Contribution à l'implantation optimisée de l'estimateur de mouvement de la norme H.264 sur plates-formes multi composants par extension de la méthode AAA

Oussama Feki

Université Paris-Est Marne la Vallée

Ecole Doctorale Mathématiques et STIC

Université de Sfax

École Doctorale Sciences et Technologies

THESE

Pour obtenir le grade de

Docteur en Informatique

Présentée et soutenue par

Oussama Feki

Contribution A L'implantation Optimisée D'estimateurs De Mouvements De La Norme H.264 Sur Plates-Formes Multi Composants Par Extension De La Méthodologie AAA

Soutenue publiquement le 13 Mai 2015 devant le jury composé de :

Président :	**Mounir Samet**	**Professeur**	**Université de Sfax (Tunisie)**
Rapporteurs :	**Mohamed Hédi BEDOUI**	**Professeur**	**Université de Monastir (Tunisie)**
	El-Bay Bourennane	**Professeur**	**Université de Bourgogne (France)**
Examinateurs :	**Serge WEBER**	**Professeur**	**Université de Lorraine (France)**
Directeurs :	**Mohamed Akil**	**Professeur**	**Université Paris-Est (France)**
	Nouri Masmoudi	**Professeur**	**Université de Sfax (Tunisie)**
Encadreur :	**Thierry Grandpierre**	**Maître de conférences**	**Université Paris-Est (France)**

Remerciement

J'aimerais avant tout exprimer ma gratitude et mes remerciements à mes directeurs de thèse, Mr. ***Nouri Masmoudi***, Professeur de l'université de Sfax et Mr. ***Mohamed Akil***, Professeur de l'université Paris-Est Marne-la-Vallée. Ce sont deux hommes d'une grande patience, très à l'écoute et très compréhensifs, et qui sont directement responsable du bon déroulement de mes travaux. Je les remercie chaleureusement pour leurs attentions et les nombreuses discussions professionnelles et personnelles que nous avons eues.

J'adresse également à mon encadrant scientifique de thèse, Mr. ***Thierry Grandpierre***, maître de conférences de l'université Paris-Est Marne-la-Vallée, mes plus sincères remerciements. J'ai énormément appris à ses côtés. Il a su me pousser à toujours faire mieux, et m'a souvent aidé à surmonter les difficultés de ce cheminement qu'est le doctorat. Encore plus que ses grandes qualités scientifiques, j'ai beaucoup apprécié ses qualités humaines, en particulier l'écoute, le partage et la compréhension.

Je remercie vivement Mr. ***Yves Sorel***, Directeur de recherche à l'INRIA, pour ses précieux conseils et le partage de ses nombreuses connaissances lors de nos réunions. Sa contribution a permis d'apporter plus de rigueur scientifique à mes travaux de thèse.

J'exprime ma gratitude à Mr. ***Mohamed Hédi Bedoui***, Professeur à la Faculté de Médecine de Monastir ainsi qu'à Mr. ***El-Bay Bourannane***, Professeur de l'université de Bourgogne pour avoir accepté de juger ces travaux en tant que rapporteurs. Je tiens à exprimer ma reconnaissance à Mr. ***Mounir Samet***, Professeur de l'université de Sfax, pour avoir accepté de présider ce jury. Merci à Mr. ***Serge Weber***, professeur de l'université de lorraine, qui a accepté de faire partie de ce jury.

Une thèse est un travail assez personnel qui s'inscrit toutefois dans une équipe. Je remercie donc tous mes collègues doctorants, tous les membres temporaires ou permanents que j'ai croisé au cours de cette expérience au Laboratoire d'Electronique et des Technologies de l'Information (***LETI***) à Sfax et à l'***ESIEE-Paris***. Ils m'ont beaucoup apporté pendant ces dernières années tant au niveau professionnel qu'au niveau personnel.

Comme il y a aussi une vie en dehors du travail. j'ai eu la chance de rencontrer hors du labo des personnes qui sont aujourd'hui des amis : Aymen, Houda, Rostom, Asma et tant d'autres ! Auxquels je dois aussi de bons moments de sincérité et de complicité.

Enfin j'exprime toute ma gratitude à ma famille : mes parents, mes frères et ma sœur, mes oncles et tantes et mes cousins et cousines, qui ont supporté avec moi les bons et les mauvais moments et qui m'ont donné la force d'aller au bout de cette aventure.

À mes parents.

À mes frères et ma sœur.

À ces personnes qu'on aime tant et qui nous ont quittés prématurément.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction générale

Contexte

Les systèmes embarqués font partie de notre quotidien. En effet, on les utilise dans tous les domaines sans mêmes nous en rendre compte. Ces systèmes embarqués sont soumis à des contraintes de temps réel, de taille, de consommation et souvent de coût, de plus en plus sévères. L'utilisation d'architectures mixtes, comportant des composants programmables et d'autres reconfigurables, est une des solutions envisageables pour satisfaire ces contraintes. Mais l'implantation et l'optimisation d'une application temps réel sur ce type d'architecture est une tâche complexe qui nécessite un long temps de conception. Ainsi, l'utilisation d'une méthodologie rigoureuse de conception et d'optimisation d'implantation des applications sur ce type d'architectures est nécessaire.

La méthodologie Adéquation Algorithme Architecture (AAA) est une des méthodologies de prototypage rapide permettant l'aide à la conception des systèmes embarqués temps réel. Elle vise l'optimisation de l'implantation des applications sur les architectures multi-composants **programmables**. Cette méthodologie couvre toutes les étapes d'implantation. En effet, elle permet la spécification de l'algorithme et de l'architecture puis le partitionnement et l'ordonnancement automatique des tâches de l'application sur les différents opérateurs et enfin la génération automatique des codes correspondants. Cette méthodologie a été étendue pour pouvoir optimiser l'implantation des applications sur les architectures **mono-FPGA** et générer automatiquement le code VHDL correspondant. Malheureusement, cette méthodologie ne couvre pas encore les architectures **mixtes** constituées de composants programmables et d'autres reconfigurables.

Objectifs de la thèse

Nous nous intéressons aux architectures mixtes à base de composants programmables (processeurs d'usage général, DSP) et de composants reconfigurables (FPGA) pour l'implantation des applications temps réel. Etant donnée l'ampleur du travail d'implantation et d'optimisation d'une telle application temps réel sur ce type d'architecture, il est important de suivre une méthodologie rigoureuse. Nous nous basons donc sur la méthodologie de prototypage rapide Adéquation Algorithme Architecture (AAA) qui vise à l'implantation optimisée d'algorithmes temps réel sur les architectures multi-composants. Cette méthodologie est basée sur un formalisme rigoureux de transformation de graphe et est implantée dans deux outils logiciels SynDEx et SynDEx-IC. SynDEx couvre les architectures programmables et SynDEX-IC les architectures mono-FPGA. Etant donné que

nous voulons couvrir les architectures mixtes, il s'agit de proposer une extension de cette méthodologie et des outils associés. Cette extension comporte quatre volets :

1. L'extension du modèle d'architecture pour pouvoir modéliser l'ensemble des composants programmables et des composants reconfigurables en tenant compte des spécificités de chaque type.
2. L'extension du modèle d'implantation de l'algorithme sur l'architecture pour l'adapter au parallélisme massif offert par l'utilisation des composants reconfigurables.
3. La création d'un nouvel outil utilisant partiellement SynDEx et SynDEx-IC pour pouvoir optimiser l'implantation des applications sur les architectures mixtes.
4. La génération automatique des communications entre les composants programmables et les composants reconfigurables et des codes correspondants à l'implémentation optimisée de l'application sur l'architecture mixte.

En guise d'exemple de validation, nous choisirons la vidéo numérique qui envahie notre quotidien. Cette vidéo doit être compressée pour faciliter son stockage et son transfert. Nous nous intéressons essentiellement à l'estimation de mouvement qui est une opération clé pour la compression vidéo, mais qui implique une complexité de calcul conséquente. Sur un encodeur vidéo, jusqu'à 60% de la charge de calcul est dédiée à cette opération. Le contexte de la haute définition et l'évolution des standards de compression vidéo (ex. MPEG-4 H.264/AVC) contribuent à accroître les contraintes pour une exécution temps-réel. L'implantation optimisée d'estimateurs de mouvement sur une architecture mixte doit alors être étudiée dans ce nouveau contexte.

Structure de la thèse

Cette thèse est composée de sept chapitres, le premier chapitre présente le contexte générale, les objectifs et la structure de la thèse.

Le second chapitre étale l'état de l'art des systèmes embarqués et des différents types de composants de traitement qu'ils peuvent contenir. Ces systèmes peuvent contenir des composants de traitement programmables, des composants de traitement reconfigurables ou combiner ces deux types de composants de traitement. Ce premier chapitre présente aussi les principaux outils permettant d'aider à la conception des systèmes embarqués. La plupart de ces outils ciblent un seul type d'architecture mais il existe aussi des outils d'aide à la conception qui ne font pas de restrictions sur le type d'architecture. De plus, ces outils diffèrent par les étapes du flot de conception qu'ils traitent (la modélisation, la simulation, l'exploration de l'espace des solutions et la génération automatique de codes). En effet, chaque outil peut couvrir une ou plusieurs étapes de ce flot de conception.

Le troisième chapitre présente la méthodologie de prototypage rapide Adéquation Algorithme Architecture (AAA). Cette méthodologie repose sur un graphe d'algorithme et un graphe d'architecture spécifiés par l'utilisateur, elle donne des règles de transformation permettant de construire un graphe d'implémentation optimisé. La méthodologie Adéquation Algorithme Architecture existe sous deux variantes : AAA pour les composants programmables et AAA pour les composants reconfigurables. La méthodologie AAA pour les composants programmables vise l'optimisation du temps d'exécution des algorithmes sur les architectures multi-composants programmables. Elle permet le partitionnement/ordonnancement automatique des opérations de l'algorithme sur les différents opérateurs de l'architecture en tenant compte des communications. De plus, cette méthodologie permet de générer le code correspondant à ce partitionnement optimisé pour

chaque opérateur. La deuxième variante de la méthodologie AAA est apparue à la suite d'une extension de cette méthodologie pour optimiser l'implémentation des algorithmes sur des architectures reconfigurables (mono-FPGA). L'optimisation s'effectue alors par une technique similaire au déroulage de boucles. Elle permet de générer un code VHDL optimisé. Ces deux variantes de la méthodologie AAA sont implémentées dans deux outils de conception assistée par ordinateur : SynDEx et SynDEx-IC qui seront également présentés.

Le quatrième chapitre présente notre extension du modèle d'architecture AAA pour pouvoir modéliser les architectures mixtes : contenant des composants programmables et d'autres reconfigurables. En effet le modèle d'architecture initial de la méthodologie ne permettait pas de tenir compte du parallélisme massif offert par l'utilisation des composants reconfigurables. Ce chapitre fait aussi un état de l'art des algorithmes de partitionnement pour le co-design. Nous y présentons aussi notre algorithme de couplage des outils SynDEx et SynDEx-IC pour l'optimisation du partitionnement et de l'implémentation des algorithmes sur les architectures mixtes. Cet algorithme permet un passage automatique du graphe d'algorithme du modèle AAA initial à notre modèle étendu. Ensuite, il utilise SynDEx d'une part pour effectuer le partitionnement/ordonnancement des opérations sur les différents composants du graphe d'architecture et SynDEx-IC d'autre part pour l'optimisation de l'implémentation des parties distribuées sur les composants reconfigurables.

Le cinquième chapitre présente la méthode de génération automatique des codes utilisés dans la méthodologie AAA. Il présente aussi les techniques pouvant être utilisées pour établir la communication entre les composants programmables et les composants reconfigurables. Comme la méthodologie AAA pour les composants reconfigurables (SynDEx-IC) vise des architectures mono-FPGA, elle ne génère pas des composants pour gérer les communications du FPGA avec d'autres composants. C'est pourquoi nous proposons une IP de communication dont l'architecture est présentée dans ce chapitre. Cette IP de communications est générée automatiquement pour exécuter une liste d'opérations de communication préétablie à partir des macros générées par SynDEx. Cette IP permet l'envoie et la réception des données mais aussi de synchroniser les opérations de communication et de traitement.

Le sixième chapitre présente l'implantation de notre algorithme de couplage des outils SynDEx et SynDEx-IC. Le langage utilisé pour programmer cet algorithme de couplage est Python. Il est choisi car c'est un langage de script orienté objet qui dispose de structures de données évoluées et de plusieurs outils de manipulation de fichiers et de chaines de caractères. La deuxième partie du sixième chapitre présente brièvement la norme de codage vidéo H.264 puis en détails l'estimation de mouvement qui est utilisée comme application pour valider nos travaux. Nous présentons les résultats de partitionnement et d'optimisation d'implémentation donnés par notre outil de co-design pour l'implémentation de l'estimateur de mouvement de la norme H.264 sur une architecture composée d'un processeur NIOS II et un FPGA stratix III.

Nous présentons à la fin de ce document nos conclusions et des perspectives de travaux futurs.

Chapitre 2

Etat de l'art des systèmes embarqués

2.1 Introduction

Un système embarqué est un système électronique qui réalise une ou plusieurs tâches spécifiques sous contraintes de temps, taille, consommation et souvent de coût. Il fait partie d'un autre système souvent réagissant avec le monde extérieur à travers des capteurs et des actionneurs.

Les systèmes embarqués sont utilisés dans presque tous les domaines de la vie quotidienne avec des contraintes plus ou moins sévères. Par exemple les systèmes embarqués d'une montre digitale ou de contrôle d'appareils ménagers ne nécessitent pas de grands moyens de calcul, alors que les traitements multimédia comme la compression vidéo et la traduction automatique de la parole nécessitent une énorme capacité de calcul. De la même façon, les autres contraintes auxquelles sont soumis ces systèmes varient d'une application à une autre. Le concepteur est confronté au dilemme de satisfaire toutes ces contraintes, qui sont la plus part du temps contradictoires, tout en minimisant le coût et en garantissant la sécurité de fonctionnement.

Ces systèmes embarqués sont constitués de composants de calcul, de composants de mémorisation et de composants de communication.

Les composants de calcul effectuent le traitement des données. L'algorithme qu'ils exécutent peut être figé par le fabriquant (ASIC) ou spécifié par l'utilisateur (micro processeur ou FPGA).

Les composants de mémorisation ou mémoires sont des ressources matérielles permettant de stocker pour des durées et des usages variables, les données à traiter, les résultats temporaires, ainsi que des informations de configuration telles que instructions pour les processeurs. Elles peuvent être classifiées selon l'ordre d'accès aux données qu'elles contiennent en mémoires à accès séquentiel et mémoires à accès aléatoire. L'adressage des mémoires à accès séquentiel est implicite : l'accès aux données dépend de l'ordre dans lequel elles ont été enregistrées. En effet, ce type de mémoires est réalisé par un registre à décalage et l'utilisateur peut accéder soit au premier étage, soit au dernier. Ce type de mémoires est utilisé pour l'implémentation des piles et des files. Les mémoires à accès aléatoire sont équipées de bus adresse pour pointer un des registres qui la constitue. De cette façon la lecture et l'écriture des données sont indépendantes. Ce genre de mémoire est utilisé pour la sauvegarde des données à traiter, des résultats et des instructions.

Les composants de communication relient les différents composants du système entre eux. Ils implémentent les fonctions de transport et de routage des données et la fonction d'arbitrage des priorités d'accès aux ressources matérielles.

Nous allons nous intéresser essentiellement dans la prochaine section aux composants de calcul dont l'algorithme exécuté est spécifié par l'utilisateur. Ils peuvent être programmables (micro processeur) ou reconfigurable (FPGA).

2.2 Architectures matérielles de traitement

2.2.1 Architectures programmables

Malgré le jeune âge de l'industrie des systèmes informatiques, les microprocesseurs ont connu une grande évolution. En effet, la société Intel, pionnière dans la conception et la fabrication des microprocesseurs, a donné naissance en 1971 au premier microprocesseur, le 4004, suivie en 1972 du 8008 effectuant des calculs sur 4 bits et 8 bits respectivement. Peu de temps après, Motorola a proposé la famille de microprocesseurs 6800 [Betker et al., 1997]. Ensuite l'évolution des performances des micro processeurs a suivi la loi de moore [Moore, 1965], qui observe le doublement des capacités d'intégration tous les deux ans, en doublant les fréquences de fonctionnement jusqu'en 2004. Puis à cause de la forte consommation énergétique et la difficulté de dissipation thermique, il a été nécessaire de s'orienter vers la parallélisation pour améliorer les performances des microprocesseurs. Ainsi les microprocesseurs multi-cœurs, qui ont vu le jour au début des années 2000, se sont développés pour améliorer les performances des systèmes embarqués.

Le tableau 2.1 présente une classification des microprocesseurs se basant sur leur architecture.

Tableau 2.1 – Classification des architectures de microprocesseurs

Architecture	Caractérisation	Exemples
CISC	Grand nombre d'instructions et de modes d'adressage	Intel Core i7, Intel Xeon, AMD Opteron
Multi-CISC		
RISC	Nombre d'instructions réduit mais durée d'exécution des instructions optimisée	SUN Sparc, ARM Cortex, IBM PowerPC, MIPS
Multi-RISC		
DSP	Jeu d'instructions développés en vue d'optimiser les applications de traitement de signal, par exemple multiplication et accumulation...	TI TMS320C6000, Analog Device SHARC, NXP TriMedia
Multi-DSP		
Graphique (GPU)	Jeu d'instructions conçu pour le graphisme des jeux vidéo puis étendu pour le calcul massif parallèle	Nvidia Fermi GF100, ATI Radeon

Les architectures CISC (Complex Instruction Set Computer) ont un jeu d'instructions étendu en terme de nombre d'instructions, de leurs complexités et de modes d'adressage possible.

Les architectures RISC (Reduced Instruction Set Computer) minimisent le nombre de cycles nécessaires pour exécuter une instruction. Le jeu d'instructions que peuvent exécuter ces architectures est constitué d'instructions simples à taille fixe.

La figure 2.1 montre l'architecture du microprocesseur Power 7 de IBM. Il se base sur l'architecture RISC et comporte 8 cœurs avec pour chacun 12 unités de traitement.

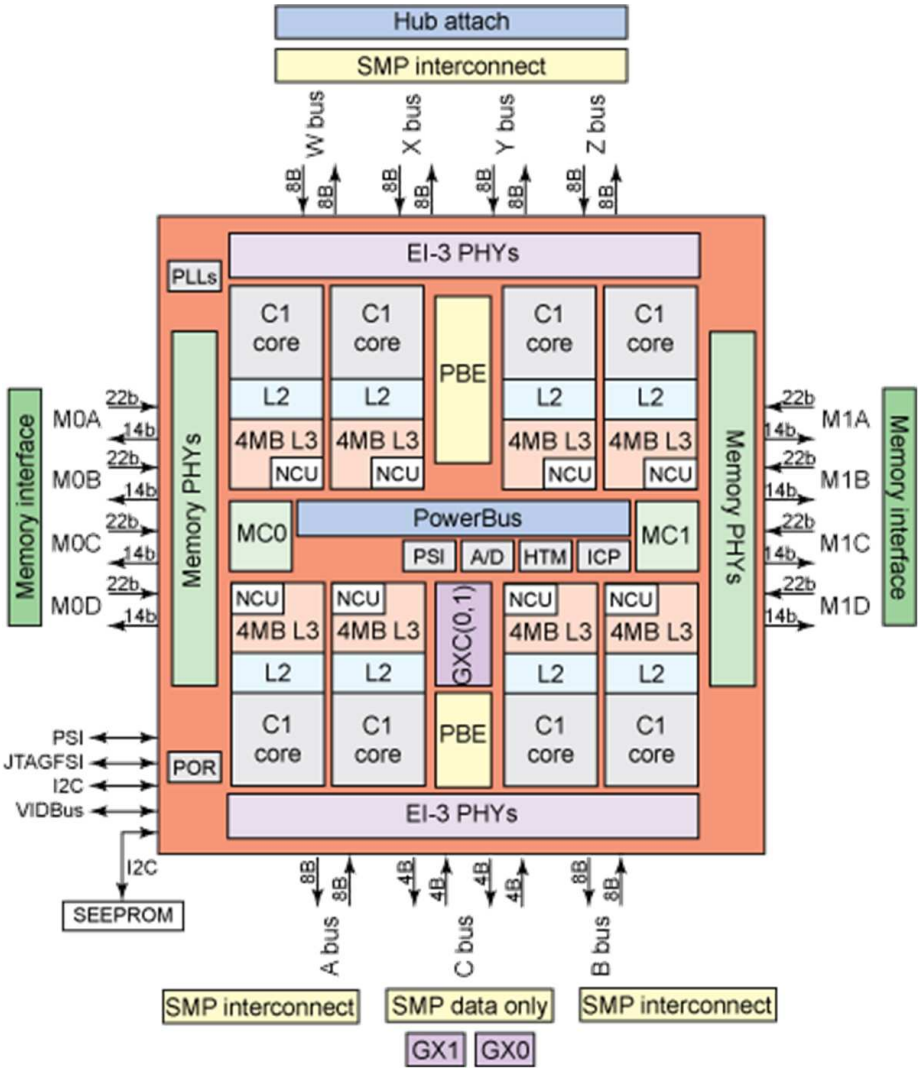

FIGURE 2.1 – Architecture du microprocesseur power 7 de IBM [Kalla et al., 2010]

Les architectures spécialisées pour le traitement de signal DSP (Digital Signal Processor) sont conçues pour répondre aux besoins des applications de traitement de signaux. En effet, ces architectures sont optimisées pour exécuter les sommes de produits. Elles disposent pour cela d'instructions spécifiques (Multiply and Accumulate MAC) reposant pour la plupart sur des multiplieurs câblés. Ces architectures permettent ainsi l'exécution d'opérations multiples par cycle. Elles sont dotées de mémoires locales aux processeurs et de module DMA (Direct Memory Access) permettant d'optimiser le temps d'accès aux mémoires externes et aux périphériques d'acquisition. La figure 2.2 représente l'architecture fonctionnelle du DSP TMSC6670 de Texas Instruments.

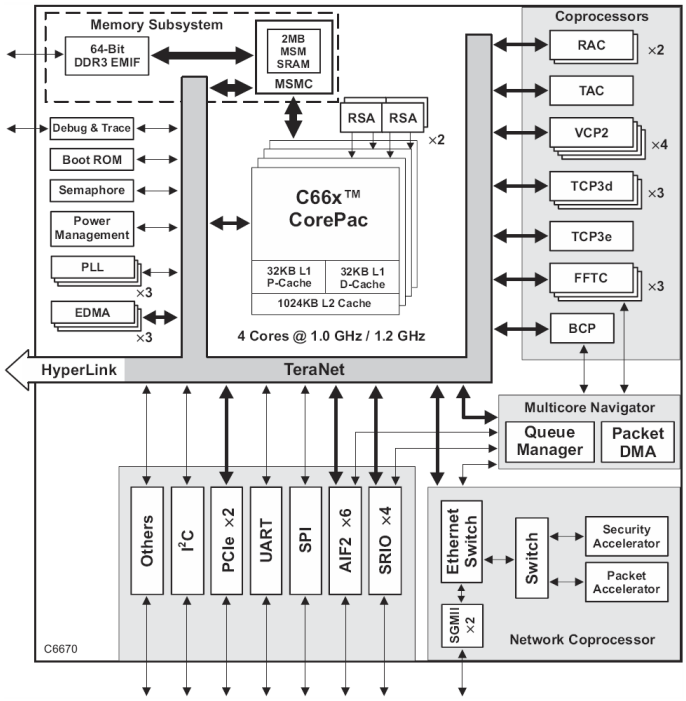

FIGURE 2.2 – Diagramme fonctionnel du TMS320C6670 de TI [Instruments, 2012]

Ce DSP comporte 4 cœurs C66x (figure 2.3) dont chacun contient 8 unités de traitement réparties sur deux chemins de données identiques. Les unités de chaque chemin de données sont .D, .S, .L et .M. Toutes ces unités peuvent fonctionner en parallèle. Ainsi chaque cœur C66x peut exécuter jusqu'à 8 instructions par cycle et on peut atteindre un degré élevé de parallélisme.

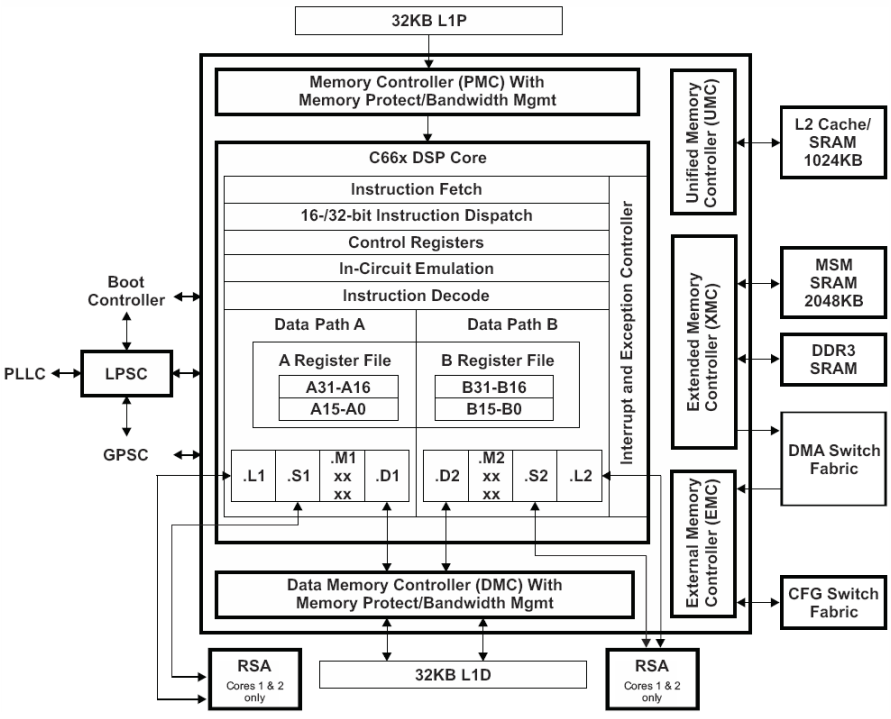

FIGURE 2.3 – Diagramme fonctionnel du cœur C66x de TI [Instruments, 2012]

Les processeurs graphiques GPU (Graphic Processing Unit) sont dédiés aux calculs de rendu 3D : transformation géométrique, placage de texture [Mcclanahan, 2010]. Les premiers processeurs graphiques étaient équipés d'un pipeline matériel qui ne pouvait pas être programmé. Ils manquaient donc de flexibilité, bien qu'ils ont accéléré considérablement les rendus graphiques. Au début du 21ème siècle, les premiers GPU complètement programmables sont apparus afin d'offrir de nouveaux effets 3D. Les GPU sont devenus plus populaires avec l'augmentation de la demande pour les applications graphiques surtout dans le domaine des jeux vidéo pour lesquels ils ont été conçu. Mais peu à peu, cette flexibilité de programmation s'accroissant, ils sont sortis du secteur de la 3D pour être utilisés dans des applications plus généralistes ; d'où le nom de General Purpose GPU. La puissance des GPU repose sur le parallélisme massif de traitement en multipliant le nombre de cœurs de calcul. Ils permettent l'exécution en parallèle du même code sur des données différentes par plusieurs centaines de cœurs. L'efficacité des GPU est aussi rendue possible grâce aux systèmes mémoires à bande passante énorme dont ils disposent. Cependant, pour que l'implantation des algorithmes soit efficace sur ce type d'architecture, ils doivent être très réguliers et renferment un grand parallélisme de données.

La figure 2.4 représente l'architecture de la plateforme kepler-GK110 de Nvidia. Cette architecture comporte six contrôleurs mémoire de 64-bits et peut contenir jusqu'à 15 multiprocesseurs (SMX). Chaque multiprocesseur SMX contient 192 cœurs CUDA, 64 unités de calcul double précision, 32 unités de fonction spéciales et 32 unités de chargement/enregistrement.

FIGURE 2.4 – Architecture de la plateforme Kepler-GK110 et du processeur SMX de Nvidia [NVIDIA, 2012]

2.2.2 Architectures reconfigurables

Les composants de calcul des systèmes embarqués peuvent être des circuits intégrés conçus spécifiquement pour une application donnée (ASIC). Leur spécificité à une application leur permet d'atteindre de hautes performances pour une faible consommation d'énergie si l'on compare aux composants programmables qui sont généralistes.

Cependant, développer un ASIC est extrêmement coûteux et long. L'utilisation d'ASIC ne peut être amortie que par la fabrication de composants en très grande série.

Les composants reconfigurables offrent un compromis entre la flexibilité des microprocesseurs et les performances des ASIC. Les premiers composants reconfigurables à apparaitre sont les PLA (Programmable Logic Arrays). Le principe de ces circuits est présenté par la figure 2.5. Ces composants sont formés de deux plans : le plan ET et le plan OU. Le plan ET est composé de fils croisés sans contact avant configuration : les fils connectés aux entrées (et leurs complémentaires) et les fils connectés aux entrées des ports ET. De la même façon, le plan OU est formé de fils qui se croisent (sans contact avant configuration) : des fils connectés aux sorties des ports ET et des fils connectés aux entrées des ports OU. L'opération de configuration vise à créer des contacts entre les fils pour obtenir la fonction voulue.

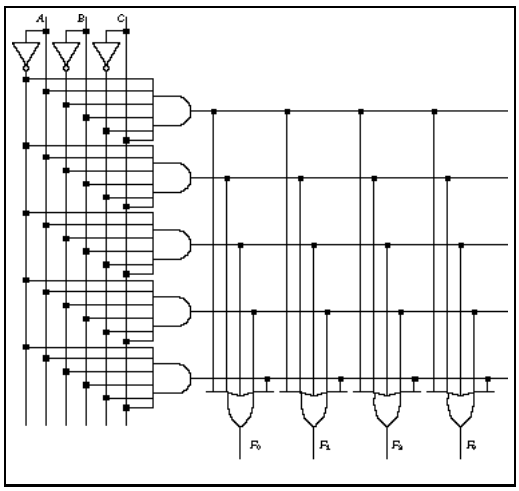

FIGURE 2.5 – Principe des PLA [Katz, 1994]

Les PLD ont évolué pour donner naissance aux circuits logiques programmables complexes (CPLD). Ces CPLD sont formés par des PLA inter-connectées entre elles par un réseau configurable. Ils comportent aussi des bascules sur les entrées/sorties du circuit.

Les FPGA (Field Programmable Gate Arrays ou "réseaux logiques programmables") sont des composants reconfigurables plus avancés que les CPLD. Un FPGA est plus flexible qu'un CPLD, il permet la mises en œuvre de circuits exécutant des fonctionnalités plus complexes, et peut être utilisé pour la mise en œuvre de circuits numériques qui utilisent l'équivalent de plusieurs millions de portes logiques. En effet, ils sont constitués d'une matrice de blocs logiques programmables, permettant de réaliser des fonctions combinatoires et des fonctions séquentielles, entourées de blocs d'entrée/sortie.

A l'échelle internationale, il existe plusieurs firmes de fabricants de circuits logiques reconfigurables à savoir : la société Xilinx, la société Altera, la société Actel ....

Les premiers FPGA modernes ont été proposés par Xilinx en 1984 [Kuon et al., 2008]. Ensuite, en fin des années 90 les familles Virtex et Spartan ont été lancées. Puis les

familles Spartan-II en 2000, Virtex-II de 2001 à 2003, Spartan-3 en 2003, Virtex-4 en 2004, Virtex-5 de 2006 à 2008, Virtex-6 et Spartan-6 en 2009 et Virtex-7 en 2010.

L'autre grand fabriquant des FPGA, Altera, a attendu 2002 pour produire les familles Stratix et Cyclone, puis Stratix II et Cyclone II en 2004, Stratix III en 2006, Cyclone III et Arria en 2007, Stratix iV en 2008, Arria II et Cyclone IV en 2009, Stratix V en 2010 et Cyclone V et Arria V en 2011.

En guise d'exemple, la figure 2.6 représente le schéma bloc du module logique adaptatif (ALM) utilisé pour implémenter les fonctions logiques dans les FPGA Stratix II d'Altera. Ces ALM sont constitués par une "Look Up Table" (LUT) à 8 entrées qui peut implémenter une ou deux fonctions, deux additionneurs et deux registres.

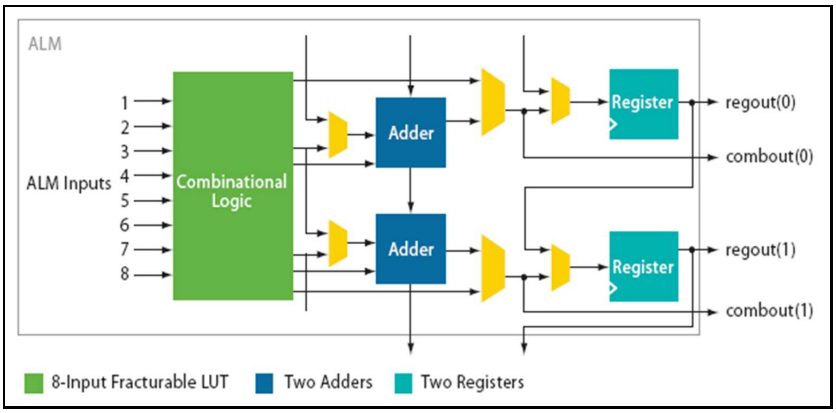

FIGURE 2.6 – Schéma bloc du module logique adaptatif [Altera, 2006]

Le bloc logique du FPGA Virtex-5 de Xilinx est présenté dans la figure 2.7. Il comporte essentiellement une LUT à 6 entrées, des multiplexeurs et une bascule. La famille Vitex-5 peut contenir jusqu'à 330000 blocs logiques [Xilinx, 2009a].

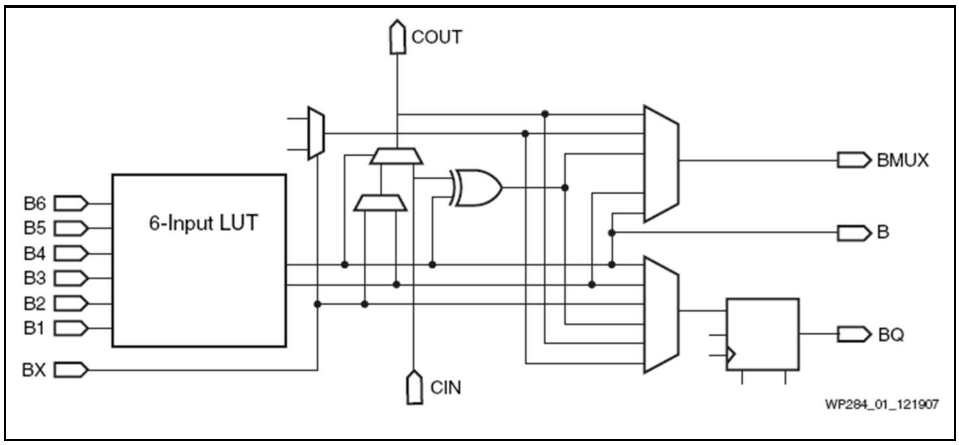

FIGURE 2.7 – Architecture du bloc logique du Virtex5 de Xilinx [Xilinx, 2007]

L'ensemble est relié par un réseau d'interconnexions qu'on peut reconfigurer pour supprimer ou ajouter des connexions entre les différents blocs logiques. On peut aussi y trouver des blocs mémoires et des opérateurs optimisés pour effectuer des opérations spécifiques. La figure 2.8 représente la structure simplifiée d'un FPGA Stratix d'altera. Sur cette figure on distingue les blocs logiques (LABs), les blocs d'entrée/sortie (IOBs), les blocs DSP pour effectuer des multiplications accumulations et des blocs mémoires.

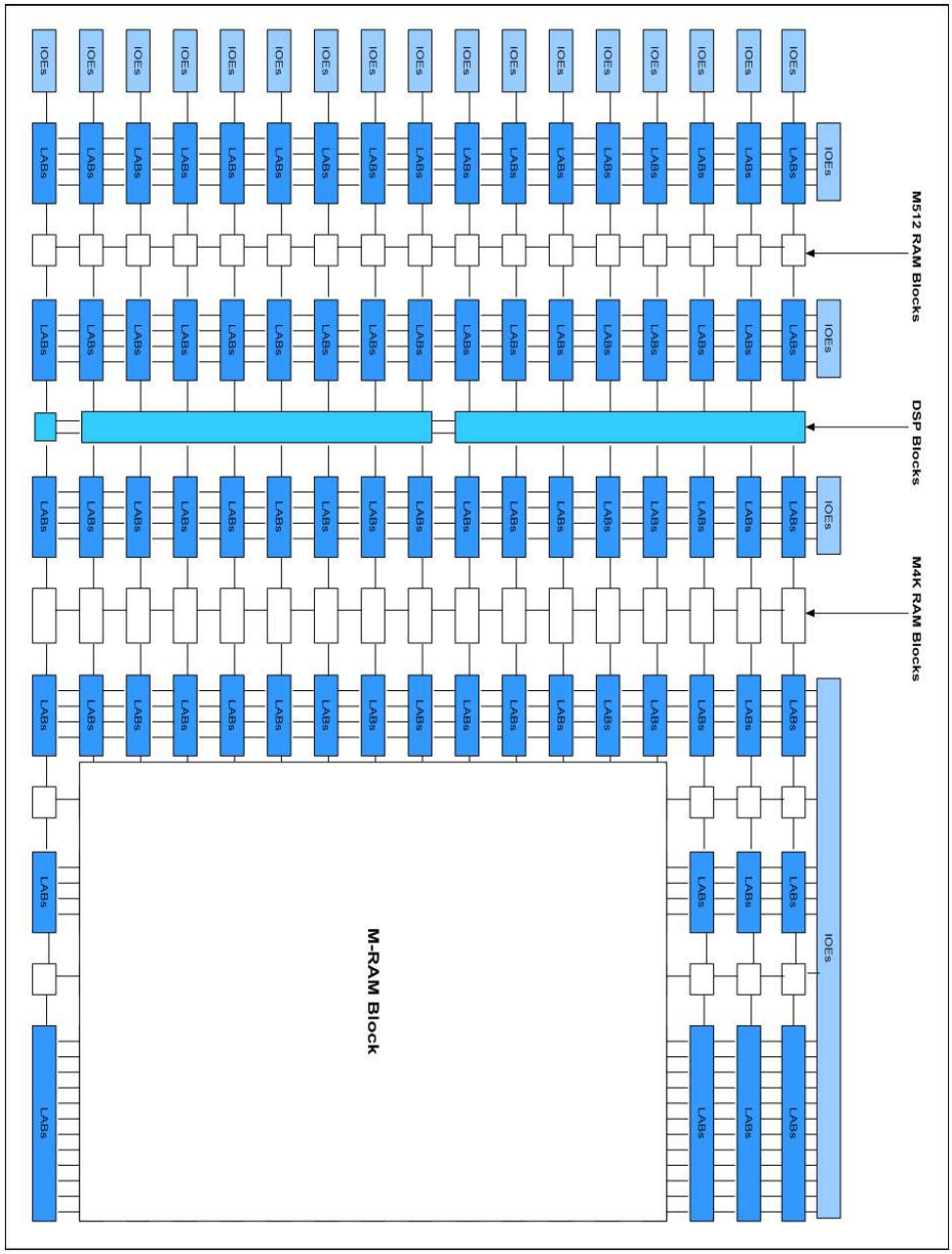

FIGURE 2.8 – Représentation du FPGA Stratix d'Altera.

Assurément, profitant du parallélisme matériel, les FPGA dépassent la puissance de calcul des processeurs en cassant le paradigme de l'exécution séquentielle. De plus, contrôler les entrées/sorties au niveau du matériel fournit des temps de réponse plus rapides et des fonctionnalités spécialisées pour correspondre étroitement aux exigences de l'application. Les frais d'ingénierie de la conception ASIC dépasse de loin ceux des solutions matérielles basées sur FPGA. L'investissement initial important dans la conception des ASIC est facile à justifier pour des prévisions de vente de milliers de puces par an, mais de nombreux utilisateurs ont besoin des fonctionnalités matérielles personnalisées pour les dizaines à des centaines de systèmes en développement. La nature même de silicium programmable signifie que vous n'avez pas de frais de fabrication ou de longs délais pour l'assemblage. Parce que les exigences du système changent souvent au fil du temps, le coût des modifications des implémentations sur FPGA est négligeable par rapport à la grande

charge nécessaire pour un ASIC.

De plus les FPGA ont la même adaptabilité de logiciels s'exécutant sur un système à base de processeur, avec l'avantage qu'on n'est pas limité par le nombre de noyaux de traitement disponibles. En effet, contrairement aux processeurs, les FPGA sont vraiment parallèles par nature, de sorte que les différentes opérations de traitement n'ont pas à concourir pour les mêmes ressources. Chaque tâche indépendante de traitement est affectée à une section spécifique du FPGA, et peut fonctionner de manière autonome, sans interférer avec les autres blocs logiques.

La technologie FPGA offre aussi une flexibilité et des capacités de prototypage rapide dans une perspective de réduction de temps de mise sur le marché. Elle permet de tester une idée ou un concept d'implémentation matérielle sans passer par le long processus de la fabrication des ASIC. On peut ensuite mettre en œuvre des changements progressifs et itérer sur une conception FPGA en quelques heures au lieu de semaines. La disponibilité croissante d'outils logiciels de haut niveau diminue la courbe d'apprentissage avec des niveaux d'abstraction plus élevés et offre souvent des cœurs IP précieux (fonctions prédéfinies) pour le contrôle avancé et traitement du signal.

Comme mentionné précédemment, les circuits FPGA sont sur le terrain évolutif et ne nécessitent pas le temps et les dépenses nécessaires à la refonte ASIC. Les protocoles de communication numérique, par exemple, ont des spécifications qui peuvent changer au fil du temps, et des interfaces basées sur les ASIC peuvent causer des problèmes de maintenance et de compatibilité. Etant reconfigurables, les circuits FPGA peuvent suivre les futures modifications qui pourraient être nécessaires.

Le flot de conception des composants reconfigurables est présenté dans la figure 2.9. Il commence par la spécification du circuit logique en utilisant un langage de description matérielle ou un outil de spécification graphique. Puis une simulation fonctionnelle est effectuée pour vérifier l'exactitude des résultats fournis par le composant spécifié. Cette première simulation ne tient pas compte des aspects temporels du circuit. Si les résultats de la simulation fonctionnelle sont faux, on doit refaire la spécification du circuit. Sinon on passe à la simulation temporelle qui vérifie que le circuit spécifié donne le résultat voulu à l'instant voulu. Puis on passe à la configuration du FPGA.

Les outils permettant l'exécution de ce flot de conception sont : ISE, VIVADO de Xilinx, Quartus II d'Altera, Libero d'Actel ....

2.2.3 Architectures mixtes : programmables et reconfigurables

L'implémentation des applications de traitement de signal sur des architectures programmables peut mener à un système de faible coût mais qui ne satisfait pas toujours les exigences en performances de ces applications. Par contre l'implémentation matérielle ou sur une architecture reconfigurable de ces applications permet une meilleur exploitation du parallélisme de l'application mais avec en contrepartie une forte augmentation du coût de développement et du coût financier. Un compromis entre performances et coût peut être obtenu en mixant les architectures programmables et les architectures reconfigurables : de telles architectures seront appelées **architectures mixtes** dans la suite de ce document. Il existe deux possibilités pour réaliser la partie programmable de ces architectures mixtes : elle peut être constituée par un ou plusieurs microprocesseurs ou peut être constituée par un ou plusieurs processeurs contenus dans les composants reconfigurables comme expliqué ci-dessous. Les différentes familles de microprocesseurs ayant été présentées dans la section 2.2.1, nous allons maintenant présenter les différents types

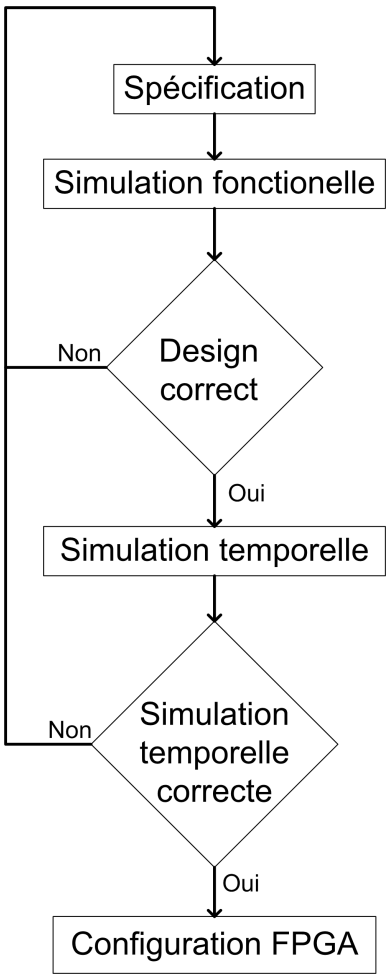

FIGURE 2.9 – Flot de conception des composants reconfigurables

de processeurs que peuvent contenir les composants reconfigurables.

Ces processeurs peuvent être câblés dans le FPGA, on parle alors de processeur **hardcore**, ou ils peuvent exister sous forme de description matérielle en langage VHDL ou Verilog, on parle ainsi de processeur **softcore**.

Les processeurs hardcores sont contenus dans les FPGA en plus des éléments standards qu'ils contiennent (blocs logiques, blocs de mémoires,...). Les processeurs hardcores les plus utilisés, aujourd'hui, sont le processeur PowerPC utilisé par exemple dans les FPGA Virtex-4 [Xilinx, 2008] et Virtex-5 [Xilinx, 2009b] de Xilinx et le processeur ARM dans le FPGA Cyclone V [Altera, 2014b] d'Altera ou la plate-forme Zynq de Xilinx [Xilinx, 2013b] par exemple. Les points positifs de l'utilisation de processeurs hardcores sont que les ressources du FPGA restent disponibles pour les accélérateurs matériels et la fréquence de fonctionnement du processeur est relativement élevée.

Les processeurs softcores sont décrits en langage de description matériel (VHDL/Verilog) utilisé d'abord pour décrire le circuit que contiendra le FPGA après configuration. Il existe des processeurs softcores libres qui peuvent être implantés dans n'importe quel circuit FPGA (LEON3, MIPs, ...). D'autres sont propriétaires et ne peuvent être implémentés que dans le circuit FPGA pour lequel il est conçu (Nios II d'Altera et MicroBlaze/PicoBlaze de Xilinx). Leur point fort réside dans leur grande flexibilité. En effet, les processeurs softcores sont flexibles et peuvent être personnalisés pour une application spécifique avec une facilité relative. On peut ainsi modifier de nombreux paramètres

comme la taille des mémoires caches, le nombre et les fonctions des unités (présence ou non d'unités DSP ou de calcul flottant ...). Par exemple, le processeur Nios II peut avoir jusqu'à 256 instructions personnalisées [Tong et al., 2006].

Les processeurs hardcore offrent de bonnes performances au détriment de la flexibilité. Les processeurs softcores fonctionnent à une fréquences faibles par rapport à la fréquence de fonctionnement des processeurs hardcores. De plus, ces processeurs consomment une partie plus ou moins grande (selon la configuration choisie et le nombre d'instructions personnalisées) des composants reconfigurables.

Les leaders du marché des circuits reconfigurables proposent des circuits mixtes basés sur leur FPGA respectifs et le processeur ARM [Corporation, 2009]. Ainsi Altera propose aujourd'hui les plateformes *Altera Soc* [Altera, 2013a] et Xilinx propose les plateformes *Zynq* [Xilinx, 2013b].

Comme tous systèmes multicomposants, les architectures mixtes nécessitent l'établissement de communications entre leurs composants [Staunstrup and Wolf, 1997].

1. Le système de communication le plus simple à mettre en place est la liaison point à point. Il permet de connecter deux composants et ne nécessite pas de système d'adressage ou de routage. La communication à travers une liaison point à point est séquentielle et implique que l'ordre de réception et d'envoi des données soit le même. Ces liaisons peuvent contenir des mémoires FIFO, on les appelle SAM : Sequential Acces Memory [Grandpierre, 2000][Grandpierre and Sorel, 2003].
2. L'utilisation de bus pour les communications permet de connecter plusieurs composants entre eux. Le bus permet d'établir à chaque instant une liaison point à point entre un couple de composants, un maitre qui gère la communication et un esclave, qui lui sont connectés. Il permet aussi la diffusion matérielle des données, c'est à dire envoyer une donnée à plusieurs récepteurs en même temps. La communication par bus nécessite l'utilisation d'un système d'arbitrage pour gérer l'utilisation du bus et définir les priorités entre les différents composants du système. Les performances d'un tel système de communication décroissent exponentiellement avec le nombre de maitres qui peuvent gérer les communications. La figure 2.10 montre une représentation simplifiée d'un système basé sur un bus. Le bus permet de connecter les différentes unités de traitement (processeurs ou circuits câblés) entre elles mais aussi aux mémoires et périphériques du système.

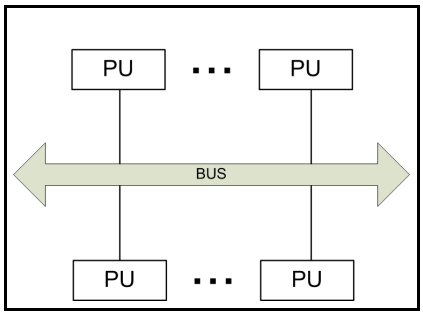

FIGURE 2.10 – Représentation d'un système de communication par bus

3. La communication par mémoire partagée ne définie pas un nouveau canal physique de communication mais se base sur une liaison point à point ou à travers un bus. En effet, ce mécanisme de communication nécessite une liaison entre l'émetteur et une mémoire et entre cette mémoire et le récepteur. Ce type de communication peut

utiliser des RAM double, triple ou quadruple ports (les bus d'adresse, de contrôle et de données sont multipliés). Mais elles sont alors de petite taille. Sinon il faut un arbitre pour gérer l'accès à la RAM [Grandpierre, 2000]. La communication s'effectue en deux étapes : l'émetteur écrit la donnée à transférer dans la mémoire partagée puis le récepteur accède à cette mémoire pour lire cette donnée. L'utilisation d'une mémoire partagée augmente relativement le temps de communication car les accès mémoire sont multiples et lents. La communication par mémoire partagée est simple à mettre en place car elle implique de simples opérations d'écriture et de lecture dans une mémoire et ne nécessite pas toujours une synchronisation. La figure 2.11 montre un système de communication basé sur une mémoire partagée. Les canaux 1 et 2 peuvent être des liaisons point à point ou un ou plusieurs bus.

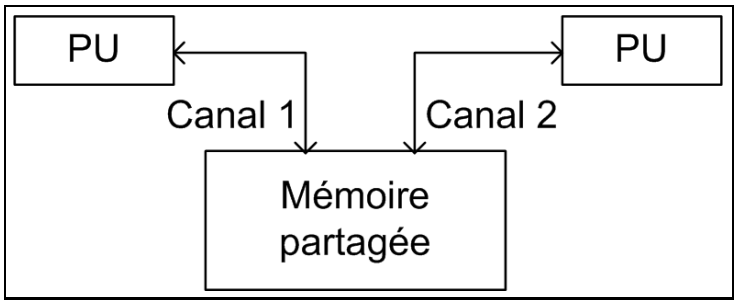

FIGURE 2.11 – Représentation d'un système de communication par mémoire partagée

4. Les liaisons point à point sont la brique de base des réseaux sur puces (NOC) [Bjerregaard and Mahadevan, 2006]. En effet, un réseau sur puce est constitué par des routeurs connectés entre eux à travers des liaisons point à point. Ces routeurs peuvent être intégrés dans les composants de traitement ou bien former des composants à part connectés aux composants de traitement par des liaisons point à point. Les performances (bande passante) de ces réseaux sur puces sont étroitement liées à la topologie du réseau mais sont généralement meilleures que les autres moyens de communication (comme la communication par mémoire partagée ci dessus) surtout pour les systèmes contenant un grand nombre de composants. Cependant, les systèmes de routage de ces réseaux peuvent être complexes et nécessiter un coût et un temps de développement assez important. La figure 2.12 représente un réseau sur puce en maille. Ce système comporte 6 routeurs connectés chacun à une unité de traitement (PU) qui peut être un processeur et sa mémoire ou un circuit câblé pour effectuer un traitement quelconque.

Si le canal de communication ne comporte pas de système de mémorisation (buffer ou mémoire), l'émetteur et le récepteur doivent être actifs en même temps et participer à la communication. Donc une synchronisation est nécessaire.

2.2.4 Synthèse des systèmes temps réel embarqués

Le choix du système utilisé pour l'implémentation des applications temps réel embarquées dépend des contraintes auxquelles sont soumises ces applications : performances, mais aussi flexibilité, coûts matériel et de développement. Le tableau suivant présente une comparaison entre ces différents types d'architectures.

On peut constater que les architectures programmables ont un coût matériel fixe. En effet, leurs performances sont proportionnelles à leurs fréquences de fonctionnement. Mais

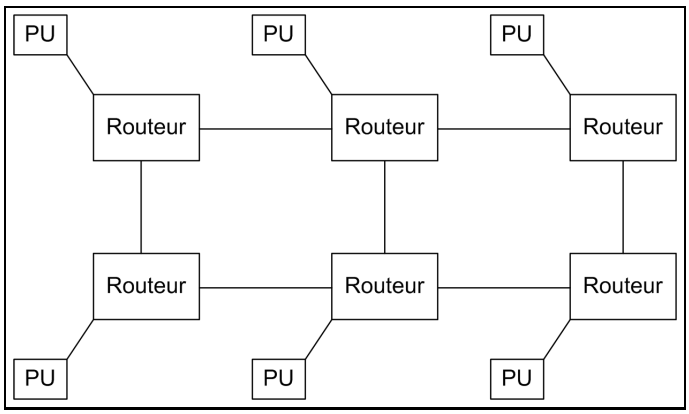

FIGURE 2.12 – Représentation d'un réseau sur puce en grille

Tableau 2.2 – Comparaison des systèmes embarqués

Type d'architecture	Flexibilité	Coût matériel	Coût de développement	Performances
Architectures programmables	Architectures pas flexibles	coût fixe	coût faible	Performances dépendant de la fréquence de fonctionnement et du nombre de composants fonctionnant en parallèle
Architectures reconfugurables	flexibilité d'architecture maximale	coût dépendant de la surface utilisée	coût élevé	Fréquences de fonctionnement relativement faibles mais grande possibilité de parallélisation
Architectures mixtes	Architecture partiellement flexible	coût variable selon la surface de la partie reconfugurable utilisée	coût très élevé	bonnes performances

l'augmentation de ces fréquences de fonctionnement induit une augmentation exponentielle de la consommation d'énergie. Donc, les concepteurs des microprocesseurs optent de plus en plus vers l'utilisation de plusieurs composants programmables fonctionnant en parallèles à des fréquences moins élevées. Ainsi, ils obtiennent une amélioration des performances tout en gardant la consommation d'énergie et l'échauffement à un niveau acceptable. Nous sommes ainsi passés à l'ère des microprocesseurs multi-cœurs.

Le point fort des architectures reconfigurables est leur flexibilité. En effet un composant reconfigurable peut être configuré pour effectuer n'importe quelle fonction numérique. Et malgré leurs fréquences de fonctionnement relativement faibles par rapport à celles des composants programmables, les architectures reconfigurables peuvent satisfaire les contraintes temps-réel de plusieurs applications grâce au parallélisme qu'elles offrent.

Les architectures mixtes représentent un compromis entre les architectures programmables et les architectures reconfigurables. Elles héritent des points forts de ces deux

types d'architectures. Par contre leur mise en place est plus complexe et nécessite donc une longue durée de conception.

2.3 Outils et méthodes de développement

Plusieurs outils de conception assistée par ordinateur des systèmes embarqués existent de nos jours. Ces outils traitent une ou plusieurs étapes de la conception : la modélisation, la simulation, l'exploration de l'espace des solutions et la génération automatique de codes. Ils peuvent cibler un seul type d'architecture ou ne pas faire de restrictions sur l'architecture cible. Dans ce paragraphe, nous allons présenter et étudier les principaux outils existant aujourd'hui dans le domaine du co-design (conception conjointe).

2.3.1 SYLVA

SYLVA [Li et al., 2013] est un environnement de synthèse au niveau système pour l'implémentation matérielle des systèmes de traitement du signal. Il est développé par le département « Electronic systems » du « Royal Institute of Technology » à Kista en Suède. Il effectue une exploration de l'espace des implémentations possibles et permet de générer automatiquement une implémentation matérielle sur ASIC ou FPGA.

L'application est décrite sous forme de graphe flot de données dont les sommets sont des fonctions typiques de traitement de signal (FFT, FIR, viterbi, ...). A chacune de ces fonctions correspondent plusieurs implémentations possibles pré-conçues, vérifiées et caractérisées. Ces implémentations, appelées FIMPs (Function IMPlimentations), sont regroupées dans des bibliothèques. Elles différent par leurs latences, l'énergie qu'elles consomment et la surface qu'elles occupent.

SYLVA effectue l'exploration de l'espace des implémentations possibles en utilisant ses bibliothèques selon 3 dimensions : le choix du FIMP, le nombre des FIMPs et le niveau de pipeline entre les différentes fonctions. L'implémentation optimisée choisie minimise la fonction coût "$K_1.A + K_2.E$" où "A" représente la surface occupée par l'implémentation, "E" représente une estimation de l'énergie consommée et "K_1" et "K_2" sont des constantes arbitraires spécifiées par l'utilisateur. Elle satisfait aussi deux contraintes : une latence totale inférieure à la latence totale maximale T_{MAX} et aucun des composants utilisés (FIMPs) n'a une latence supérieure à l'intervalle d'échantillonnage maximal R_{MAX}.

Enfin, SYLVA génère automatiquement la partie contrôle (des machines à états finis) et les multiplexeurs pour gérer le fonctionnement et connecter les différents FIMPs. Cette partie contrôle est générée, dans un premier temps, dans une représentation abstraite intermédiaire. Ensuite, cette représentation abstraite intermédiaire est transformée en implémentation spécifique à la technologie cible (ASIC ou FPGA) en utilisant des outils tiers.

SYLVA ne prend compte que des architectures matérielles câblées ou reconfigurables, il ne prend pas compte des architectures programmables.

2.3.2 MATLAB/Simulink

Simulink [MathWorks, 2012] est un environnement schématique pour la simulation et la conception par modélisation. Il prend en charge la conception au niveau système, la simulation, la génération automatique de code, et le test des systèmes embarqués. Simulink est un logiciel commercial développé par « MathWorks ». Il fournit un éditeur

graphique et une bibliothèques de blocs personnalisables. Son intégration avec MATLAB, permet d'intégrer des algorithmes MATLAB dans les modèles et d'exporter les résultats de simulation à MATLAB pour une analyse ultérieure.

La bibliothèque de blocs prédéfinis, représenté par la figure 2.13, contient les blocs couramment utilisés pour modéliser un système et peut être étendue en y incorporant des blocs personnalisés dont le fonctionnement est décrit en langage MATLAB, C, Fortran ou Ada. La modélisation hiérarchique et la personnalisation des blocs permettent de représenter des systèmes complexes de façon concise et précise.

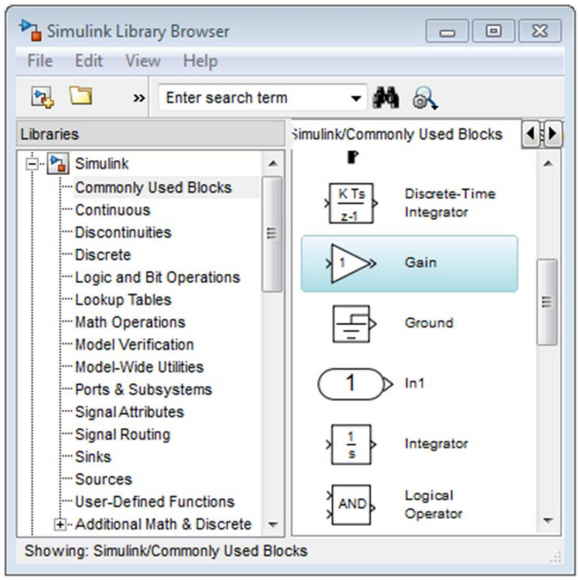

FIGURE 2.13 – Capture écran de la bibliothèque Simulink

Simulink permet de simuler le comportement dynamique du système et afficher les résultats. Il permet aussi de connecter le modèle Simulink au matériel pour le prototypage rapide, la simulation matérielle en boucle (HIL), et le déploiement sur système embarqué. En effet, grâce à un système de toolbox (extensions logicielles), Simulink fournit un support intégré pour le prototypage, le test et l'exécution de modèles sur plusieurs types de plateformes matérielles dont des DSP de Texas Instruments, des FPGA et les récentes plateformes Zynq de Xilinx. En effet, une extension logicielle spécifique permet de simuler l'exécution de l'application sur la plateforme Zynq, la génération automatique des codes s'exécutant sur le processeur et le FPGA contenus dans cette plateforme ainsi que les communications entre ces deux parties. La figure 2.14 représente une capture d'écran de l'interface de Simulink. On y distingue le menu permettant d'exécuter l'application du modèle spécifié sur la plateforme cible.

Grâce à ses extensions logicielles, les Modèles Simulink peuvent être configurés et préparés pour la génération du code. Ainsi, on peut en générer un code C, C++ ou VHDL en utilisant Simulink avec les extensions logicielles de génération de code.

Si MATLAB/Simulink permet de programmer un processeur embarqué ou un FPGA à partir d'une spécification textuelle (MATLAB) ou graphique (Simulink), il ne permet pas en revanche de générer du code pour les multiprocesseurs embarqués, les multi-FPGA ou des codes pour les architectures mixtes autre que les plateformes Zynq.

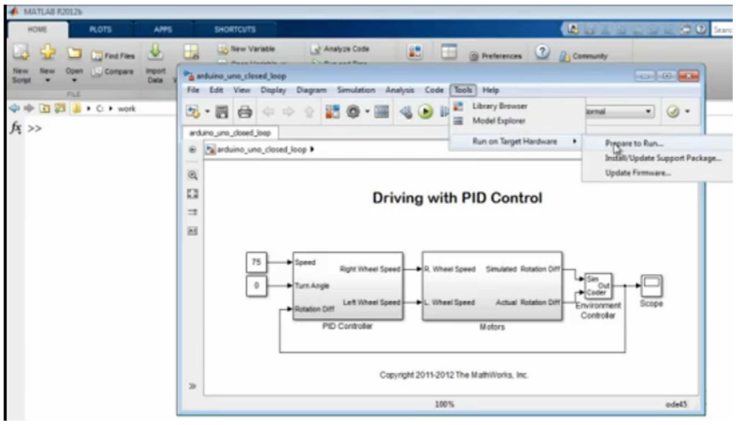

FIGURE 2.14 – Capture écran de l'interface Simulink

2.3.3 PtolemyII

Ptolemy II [Editor, 2014] est un environnement de modélisation et de simulation open source destiné à l'expérimentation de techniques de conception des systèmes, en particulier ceux qui impliquent des combinaisons de différents types de modèles. Il est développé par le « Department of Electrical Engineering and Computer Sciences » de l'université de California.

Une interface graphique (figure 2.15) est utilisée pour spécifier le modèle de l'application. Le modèle utilisé par Ptolemy II [Brooks et al., 2007] est basé sur des composants logiciels concurrents appelés "actors". Ils peuvent être atomiques (représentent le niveau le plus bas de la hiérarchie) ou composites (renfermant d'autres "actors"). Ces différents "actors" peuvent communiquer à travers les connexions entre leurs ports. Tous les "actors" (atomiques ou composites) s'exécutent en 3 étapes : mise en place (setup), répétition (iterate) et conclusion (wrapup). Durant la phase de de mise en place, les paramètres de l'actor sont initialisés, l'état local de l'actor est mis à zéro et les données initiales sont produites. Durant la phase de répétition, l'actor lit les données dans ses ports d'entrées, effectue le traitement et produit les résultats dans ses ports de sorties. Enfin, durant la phase de conclusion, l'actor libère les ressources qui lui ont été allouées durant l'exécution.

Le modèle de calcul associé à chaque actor composite est appelé domain. Il définit la sémantique de communication et l'ordre d'exécution des actors. Il se compose de 2 classes : director et receiver. Les receivers implémentent les communications. Un receiver est utilisé pour chaque canal de communication et est contenu dans le port d'entrée. Le director spécifie l'ordre d'exécution des actors.

Le langage de programmation utilisé pour la mise en place de Ptolemy II est JAVA. Ce même langage est utilisé pour spécifier le code source des composants formant le modèle. Pour améliorer les performances, le code source des actors est transformé en réduisant les appels aux méthodes. En effet, faire appel à la méthode add() pour additionner deux entiers est plus lent qu'additionner directement ces deux entiers. Ce code transformé peut-être converti en langage C pour une meilleure portabilité sur les systèmes embarqués [Tsay, 2000]. Ptolemy II permet aussi de générer un code Verilog décrivant le composant principal de la spécification de l'application [Asthana, 2010].

Ptolemy II est un outil permettant la modélisation et la simulation des applications. Malgré qu'il permet aussi la génération de code C ou Verilog, Ptolemy II ne permet pas de modéliser l'architecture cible. Donc il ne permet d'optimiser l'implémentation des applications sur les architectures mixtes.

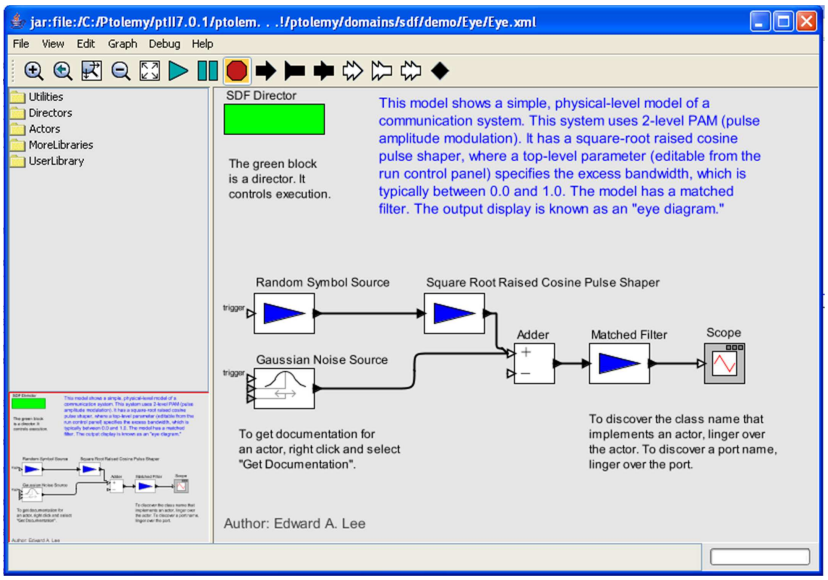

FIGURE 2.15 – Capture écran de l'interface Ptolemy II

2.3.4 CoFluent

CoFluent Studio™ [Corporation, 2012] est un outils de modélisation et simulation de systèmes embarqués composés par des processeurs. Il est développé par atlanpolitaine puis racheté par Intel. Les modèles sont saisis sous forme de diagrammes, à travers une interface graphique, en utilisant le langage optimisé spécifique à un domaine (domain-specific language DSL) de CoFluent ou en description UML standard. Pour décrire les types de données et les algorithmes, les langages ANSI C ou C++ peuvent être utilisés. Les exigences non-fonctionnelles du système ou des données de calibration des modèles telles que la durée d'exécution, la puissance sont ajoutées pour la caractérisation du modèle. Les modèles sont traduits en SystemC qui est instrumenté et génère des traces pouvant être suivis avec différents outils d'analyse. Ainsi une simulation rapide permet d'extraire les performances (comme les latences, les débits, les niveaux de tampons, les charges des ressources, la consommation d'énergie) de l'exécution du modèle établi sur des plates-formes multiprocesseurs/multi-cœurs.

CoFluent permet deux types de simulations :

- Comportementale (également appelée statistique ou à base de jetons) : les types de données et les algorithmes peuvent être laissés vides afin que les communications de données et les calculs se résument à leurs durées d'exécution. Ceci est utile pour valider les exécutions en parallèle de processus, les communications et la synchronisation inter-processus.
- Fonctionnelle : les types de données et les algorithmes correspondent aux données réelles et au traitement effectué par le système. Ils peuvent être définis dans la norme ANSI C ou C++ ou décrits avec MATLAB ou Simulink. Ce type de simulation est utile pour valider les calculs effectués par le système.

Un raffinement supplémentaire du modèle est nécessaire lors de la modélisation des applications fonctionnant sur des plates-formes multiprocesseur/multicœur avec des ressources partagées. Le concepteur du système commence par créer un modèle statique de la plate-forme cible. Ensuite, il alloue les différents processus aux cœurs, stocke les données dans les mémoires et route les communications inter-processeurs sur les liens physiques.

Cette opération est nommée allocation et est réalisée par un simple glisser-déposer. Ainsi un modèle alloué, qui peut être traduit en SystemC, est obtenu.

CoFluent ne vise donc pas directement les architectures mixtes qui nous intéressent, il s'intéresse à la modélisation de systèmes complets comme par exemple la chaine complète d'acquisition, compression, transmission, décompression et affichage de vidéo.

2.3.5 DOL

Distributed Operation Layer (DOL) [Thiele et al., 2007] est un outil permettant l'exploration de l'espace des solutions pour l'implémentation des algorithmes sur les architectures multiprocesseurs. C'est un logiciel libre développé par « l'institut fédéral de technologie de Zurich » en Suisse.

L'application est décrite, à travers une interface graphique, par deux types de spécifications : une structurelle et une fonctionnelle. La spécification structurelle est un graphe dont les nœuds représentent les processus et les arcs représentent les canaux de communication entre ces processus. Ce graphe contient des informations spécifiques à l'application comme la taille minimale des canaux, la durée d'exécution des processus .... Il est décrit en langage XML. La spécification fonctionnelle de l'application est spécifiée en langage C/C++. Chaque processus se compose de deux procédures : *init* et *fire*. La procédure *init* est appelée une seule fois pour l'initialisation du processus et l'appel de la procédure *fire* est répétitif.

La plateforme cible est décrite en langage XML. Elle décrit les ressources disponibles : les processeurs, les mémoires, les canaux de communications .... Ces éléments sont annotés chacun par ses données de performances comme le débit des bus, le délai des voies de communication, les fréquences de fonctionnement des processeurs et des bus, les tailles des mémoires, ...

L'optimisation d'implémentation de DOL est un processus itératif permettant l'exploration automatique de l'espace d'implémentations possibles. Elle part d'une implémentation spécifiée par l'utilisateur et utilise un algorithme évolutionnaire pour réduire le nombre d'implémentations à explorer.

Si DOL permet une prise en compte plus fine des processeurs de l'architecture, contrairement à Ptolemy II ou CoFluent, il ne peut pas prendre en compte les composants reconfigurables.

2.3.6 Catapult C

Catapult C est un outil commercial de génération automatique de code HDL, développé par Mentor Graphics.

L'élaboration d'un système matériel, pour répondre aux besoins complexes de nos jours, nécessite un long temps de développement. L'outil de synthèse haut nivaux Catapult permet non seulement de réduire ce temps de développement, mais aussi de réduire le risque d'erreurs. Cette réduction du temps de développement permet aux concepteurs d'essayer plusieurs solutions pour en choisir la meilleure.

L'utilisateur spécifie l'application à travers une interface textuelle en utilisant le langage C/C++ ou SystemC (figure 2.16). Catapult C permet de simuler le fonctionnement de cette spécification. L'utilisateur peut ainsi se rendre compte des erreurs de la spécification dès les premières étapes de la conception du système. Une fois cette étape passée, Catapult C peut être utilisé pour générer automatiquement le code HDL correspondant.

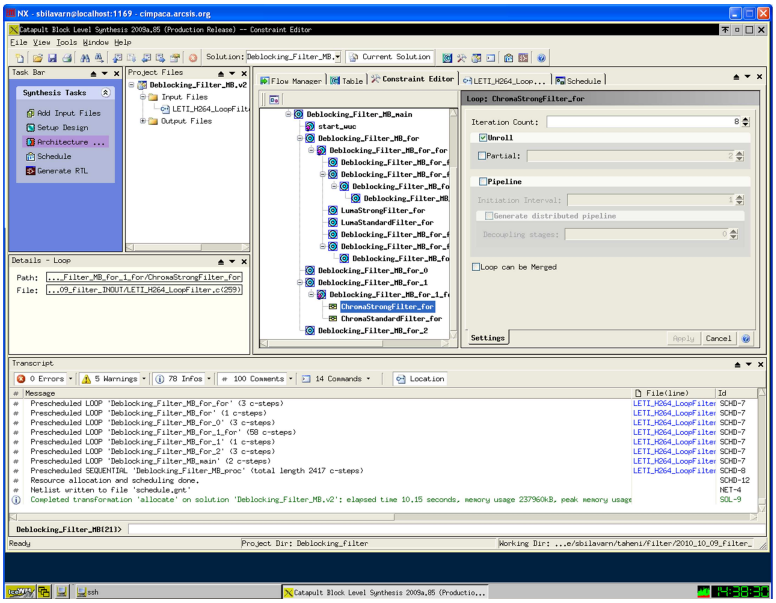

FIGURE 2.16 – Interface de Catapult C

Le code HDL est généré selon les contraintes d'utilisateur qui précisent notamment la période d'horloge, la limitation des ressources, le protocole d'entrée/sortie utilisé et le niveau de concurrence souhaité. Il est composé d'un chemin de données (les opérateurs de traitement), de composants de contrôle et des différentes connexions entre ces éléments. Catapulte C supporte plusieurs types de bus complexes pour connecter les différents composants du système. Le code généré est optimisé pour consommer une faible énergie lors de l'exécution et pour utiliser le minimum possible de mémoire.

Catapult C est donc un bon outil pour couvrir la programmation de chaque composant des architectures mixtes mais il ne permet pas la modélisation du système complet. Il ne prend pas en compte les communications et le partitionnement. C'est plutôt un outil "aval" qui peut être vu au même niveau que le compilateur.

2.3.7 Vivado HLS

Vivado® HLS [Xilinx, 2013a] est un outil commercial de génération automatique de code HDL, développé par Xilinx.

L'utilisateur spécifie l'application en langage C ou Système C en utilisant une interface textuelle (figure 2.17). Vivaldo HLS permet de simuler le fonctionnement de cette application puis de générer un code VHDL, Verilog ou Système C.

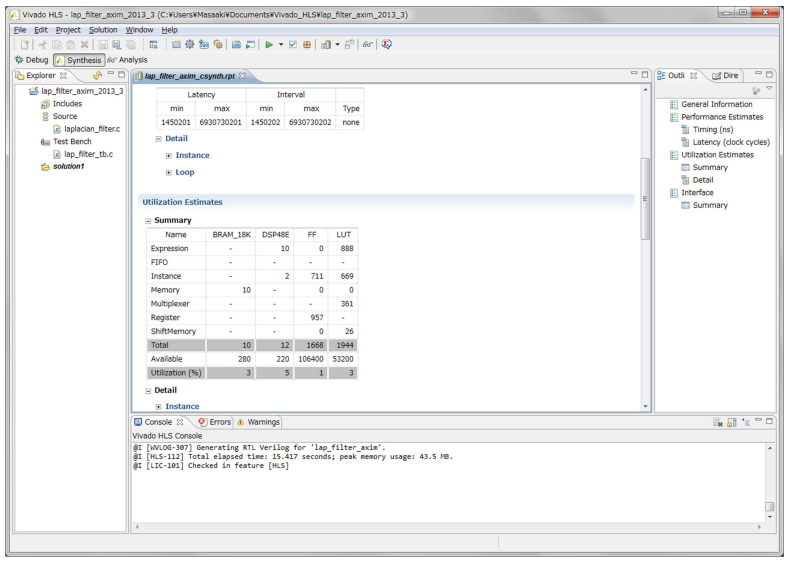

FIGURE 2.17 – Interface de Vivado HLS

Le compilateur Vivado HLS fournit un environnement de programmation similaire à ceux disponibles pour le développement d'applications sur les processeurs standards et spécialisés. La principale différence réside dans la cible de l'exécution de l'application.

Vivado HLS analyse tous les programmes en termes de :

- Opérations : les opérations réfèrent à la fois aux opérations arithmétiques et logiques d'une application qui sont impliquées dans le calcul des résultats. Avec un compilateur de processeur, l'architecture de traitement est fixe. Donc l'utilisateur ne peut affecter les performances de fonctionnement qu'en manipulant la mise en mémoire des données afin d'optimiser les performances de la mémoire cache. En revanche, Vivado HLS n'est pas limité par une plateforme fixe mais construit une plateforme spécifique de l'algorithme. Il effectue alors une analyse des dépendances de données entre les opérations et parallélise les opérations qui peuvent l'être.
- Conditionnement : Vivado HLS crée les circuits nécessaires pour l'exécution de chaque branche de conditionnement. Par conséquent, l'exécution d'un conditionnement revient à un choix de circuit plutôt qu'un changement de contexte.
- Boucles : pour réduire la latence des boucles, la première optimisation automatique appliquée par HLS est la parallélisation des opérations du corps de la boucle. La seconde optimisation est la mise en pipeline des différentes itérations. Cette optimisation nécessite une intervention de l'utilisateur, car elle dépend des débits des données d'entrées.
- Fonctions : les fonctions sont une hiérarchie de programmation qui peut contenir des opérateurs, des boucles et d'autres fonctions. Le compilateur de Vivado HLS crée un module matériel indépendant pour chaque fonction d'un niveau hiérarchique donné. Ainsi les traitements des différentes fonctions indépendantes d'un même niveau hiérarchique peuvent s'exécuter en parallèle.

Vivado HLS est un outil de génération de code HDL à partir d'un code C ou Système C. Le code généré est optimisé selon des paramètres spécifiés par l'utilisateur. Mais l'utilisation de Vivado HLS peut mener à un sur-dimensionnement des bus utilisés si la taille de données ne coïncide pas avec celle utilisée par le langage C. En effet, si par exemple le système traite des données codées sur 20 bits, ces données sont déclarées dans le code

C de type entier (32 bits). Le code HDL généré par Vivado HLS utilise des bus de taille 32 bits alors que 20 bits suffisent. Ce qui induit une occupation plus grande de surface. De plus Vivado HLS ne cible que les composants reconfigurables, donc ne prend pas en compte les parties programmables des architectures mixtes.

2.4 Conclusion

Nous nous sommes intéressés dans ce chapitre aux différentes architectures de systèmes embarqués classifiées selon les types des composants de calcul qu'ils contiennent. Ainsi les composants de calcul des systèmes embarqués peuvent avoir une architecture programmable, une architecture reconfigurable mais aussi une architecture mixte contenant des composants programmables et d'autres reconfigurables.

Nous avons présenté aussi les principaux outils utilisés pour l'aide à la conception de ces systèmes embarqués. Ces outils traitent une ou plusieurs étapes de la chaîne de conception et permettent de faciliter la conception et de réduire le temps de mise sur le marché (time-to-market) de ces systèmes.

Nous avons vu l'intérêt des architectures mixtes puisqu'elles répondent à un grand nombre de contraintes qui nous intéressent, mais nous avons aussi vu que des outils dédiés étaient nécessaires pour développer des applications sur ce type d'architecture. Comme l'état de l'art des outils l'a montré, il n'existe pas encore de tels outils, c'est pourquoi nous proposons une nouvelle méthodologie et un nouvel outil dans cette thèse. Il sera basé sur la méthodologie de prototypage rapide Adéquation Algorithme Architecture que nous allons étendre. Cette méthodologie est implémentée dans deux outils de conception assistée par ordinateur : SynDEx et SynDEx-IC que nous allons coupler.

Chapitre 3

La méthodologie AAA

3.1 Introduction

Les concepteurs de systèmes embarqués doivent tenir compte de plusieurs contraintes dont les coûts mais aussi le délais de mise sur le marché (time to market) qui deviennent stratégiques de nos jours. La réduction de ce délais permet d'avoir un produit fini avant la concurrence et peut se faire en réduisant le temps de conception et validation en utilisant le prototypage rapide.

Le prototypage rapide a pour objectif de passer de la spécification haut niveau de l'application à une implémentation temps réel sur l'architecture cible en faisant intervenir le moins possible le concepteur. Cette automatisation sert non seulement à réduire la durée de conception des systèmes complexes mais doit aussi prévenir les erreurs de conception. Nous nous intéressons dans ce chapitre à la méthodologie de prototypage rapide **Adéquation Algorithme Architecture** (AAA)[Sorel, 1994]. Cette méthodologie est choisie car elle utilise, comme nous allons voir, un modèle d'architecture permettant de modéliser la plus part des architectures matérielles, elle offre aussi la sécurité de conception. Elle utilise aussi une heuristique permettant de trouver une implémentation optimisée en un temps relativement court. Cette méthodologie existe sous deux variantes : AAA pour les circuits programmables et AAA pour les circuits reconfigurables.

Dans le prochain chapitre, nous étendrons cette méthodologie et ses outils pour couvrir les architectures mixtes qui, comme on va le voir maintenant, ne sont pas encore supportés en tant que tel.

Nous allons dans ce chapitre présenter cette méthodologie dans ces deux variantes ainsi que les outils (SynDEx pour les architectures multi-composants programmables et SynDEx-IC pour les circuits reconfigurables) se basant sur cette méthodologie.

3.2 Présentation générale de AAA

L'Adéquation Algorithme Architecture [Sorel, 1996] consiste à trouver l'implantation d'un algorithme sur une architecture en réduisant le temps d'exécution tout en utilisant le minimum de ressources matérielles possibles. Cette méthodologie est développée dans le cadre du projet OSTRE de l'INRIA et a pour but d'aider au prototypage rapide d'applications temps réel sur des architectures distribuées hétérogènes. Elle a été étendue [Kaouane, 2004] pour optimiser l'implémentation des algorithmes sur les architectures reconfigurables.

La méthodologie AAA [Grandpierre and Sorel, 2003], comme le montre la figure 3.1, utilise trois types de graphes : un graphe flot de données conditionné factorisé mettant en évidence le parallélisme potentiel de l'algorithme, un graphe d'architecture mettant en évidence le parallélisme disponible et un graphe d'implantation de l'algorithme sur l'architecture. Ce graphe d'implantation est le résultat de l'adéquation, il est obtenu par des transformations de graphes rigoureuses énoncées dans [Grandpierre, 2000]. Cette méthodologie permet ensuite la génération automatique d'exécutifs distribués temps réel.

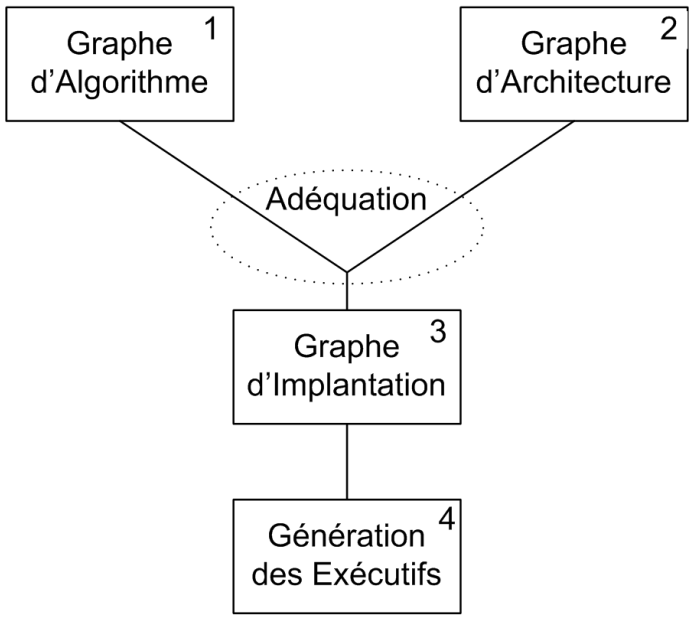

FIGURE 3.1 – Flot de conception de la méthodologie AAA

3.3 AAA pour les circuits programmables : SynDEx

3.3.1 Modèle d'algorithme

Le modèle d'algorithme constitue une des entrées fournies à la méthodologie AAA (rectangle 1 de la figure 3.1) pour pouvoir trouver une adéquation de l'algorithme sur l'architecture. L'algorithme est modélisé par un graphe factorisé conditionné de dépendance de données (O,D) où O est l'ensemble des sommets et D est l'ensemble des arcs les reliant. Chaque sommet de O représente une opération de l'algorithme. Cette opération peut être conditionnée, c'est à dire ne s'exécute que si une condition d'exécution est vérifiée, ou bien non conditionnée et s'exécute sans conditions. Chaque arc de D représente une dépendance de données ou de conditionnement entre une opération productrice et une ou plusieurs opérations consommatrices. Les sommets peuvent être de cinq types :

* Constant : un sommet constant est une abstraction d'une valeur. C'est un sommet qui n'a pas d'entrées et produit à chaque fois qu'il est exécuté cette même valeur. Ce sommet ne consomme pas de temps de calcul.
* Sensor : un sensor est un sommet représentant un capteur qui produit à l'algorithme ses données d'entrées. C'est un sommet ne possédant que des sorties. Un sommet sensor est caractérisé par une durée d'exécution qui dépend de l'opérateur qui l'exécute.

* Actuator : un actuator représente un actionneur qui consomme les données produites par l'algorithme. Il ne possède que des entrées. L'actuator est caractérisé par sa durée d'exécution qui dépend de l'opérateur qui l'exécute.
* Delay : un sommet delay, ou retard, modélise une mémorisation de données. Il doit avoir au moins une entrée et une sortie de même type. C'est l'unique façon de propager des données produites par une opération à une exécution ultérieure de l'algorithme.
* Function : les sommets functions modélisent les opérations de calcul de l'algorithme. A chaque exécution, ils consomment une donnée à chacune de leurs entrées, les transforment pour produire une donnée à chaque sortie. Chaque sommet function se caractérise par sa durée d'exécution qui dépend de l'opérateur qui l'exécute.

Les algorithmes de traitement de signal itèrent des traitements identiques sur des données différentes. Pour simplifier leurs présentations, on utilise la notion de factorisation.

La factorisation est la représentation d'un ensemble de traitements itératifs par un seul exemplaire (itération), appelé motif de répétition, en indiquant le nombre d'itérations, appelé aussi facteur de répétition. Un motif de répétition est délimité par une frontière de factorisation.

De plus, les systèmes décrits par le modèle AAA sont des systèmes temps réel réactifs qui réagissent avec leur environnement de manière discrète sous forme d'une répétition infinie d'une séquence acquisition-calcul-action. Pour modéliser cet interaction entre le système et son environnement, on définit par, analogie à la frontière de factorisation finie, la frontière de factorisation infinie. C'est une frontière de factorisation avec un facteur de factorisation infinie, et qui englobe la totalité de l'algorithme (la séquence acquisition-calcul-action) [Grandpierre, 2000].

Pour mieux expliquer la notion de factorisation, voici l'exemple du produit d'une matrice par un vecteur. Le produit M d'une matrice c de dimension 3x3 par un vecteur e de dimension 3 (figure 3.2(a)) peut se décomposer en trois produits scalaires. La figure 3.2(b) représente la décomposition de ce produit matrice-vecteur ainsi que sa factorisation (figure 3.2(c)). La frontière de factorisation est représentée par le cadre en pointillé.

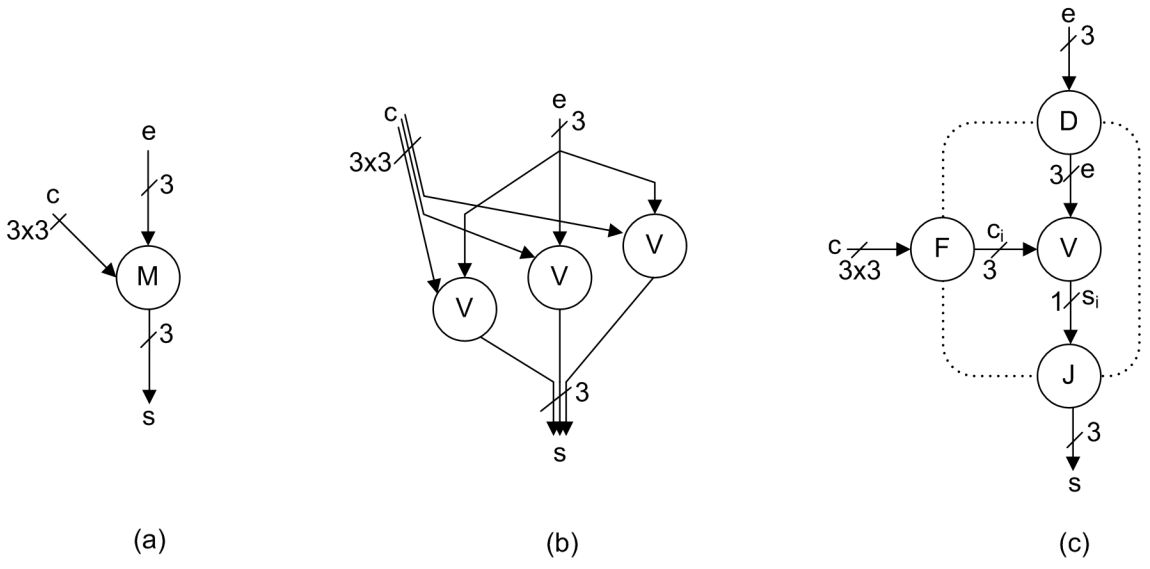

FIGURE 3.2 – Décomposition et factorisation d'un produit matrice vecteur

Cette figure 3.2(c) fait apparaitre trois nouveaux types de sommets particuliers associés à la frontière de factorisation : D pour diffusion, F pour fork et J pour Join. La figure 3.3 représente la décomposition et la factorisation du produit scalaire, c'est à dire le sommet "v" de la figure 3.2(c). La figure 3.3(c) fait apparaitre un nouveau type de sommet de frontière : le sommet I pour Iterate [Kaouane, 2004]. Pour traverser une frontière de factorisation, chaque donnée doit passer par un sommet particulier. Ces sommets servent à délimiter la frontière et à spécifier la façon dont ces données sont transmises entre l'intérieur et l'extérieur de la frontière ainsi que d'une itération à une autre.

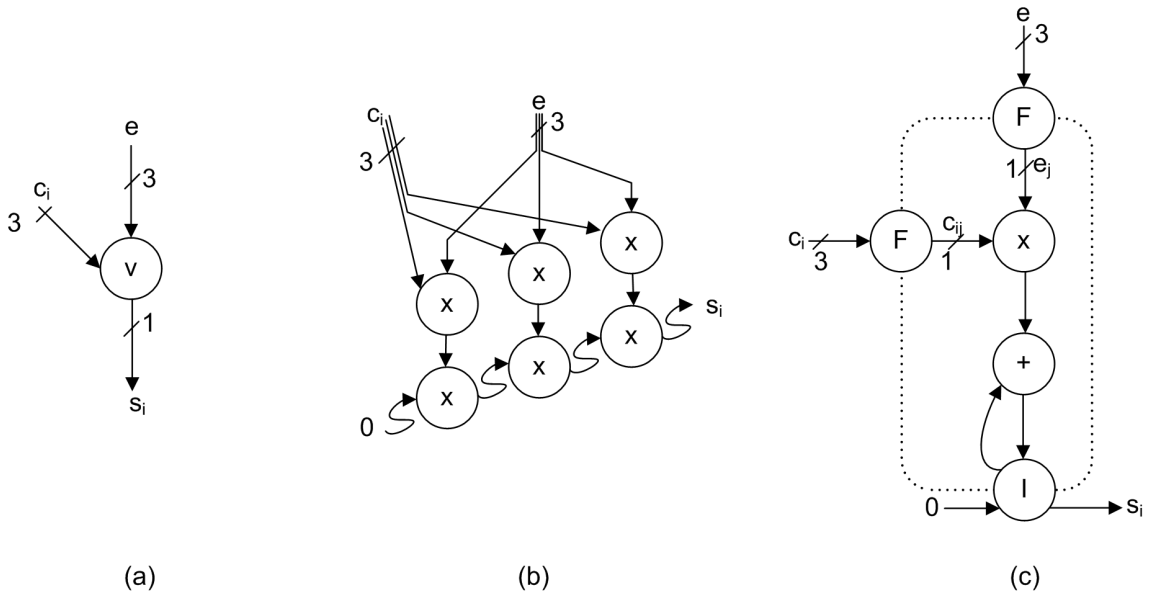

FIGURE 3.3 – Décomposition et factorisation d'un produit scalaire

Ces sommets peuvent être de quatre types :

* Sommet Fork : noté F, c'est un sommet d'entrée de la frontière. Pour une frontière qui se répéte n fois, il permet de diviser les données de taille d à son entrée en n sous-ensembles de taille d/n. On interdit le cas où d n'est pas divisible par n.
* Sommet Diffuse : noté D, c'est un sommet d'entrée de la frontière. Il reproduit à chaque itération la même donnée reçue à son entrée.
* Sommet Join noté J, c'est un sommet de sortie de la frontière. Il permet de faire l'opération inverse de celle effectuée par le Fork. En effet, un sommet Join permet de regrouper n données de dimension d/n chacune en une donnée de dimension d.
* Sommet Iterate : noté I, c'est un sommet d'entrée-sortie. Il permet de modéliser les dépendances inter-itérations.

3.3.2 Modèle d'architecture

Nous avons vu que le modèle d'algorithme AAA qui est décrit par un graphe. Nous allons maintenant voir que l'architecture est aussi décrite par un graphe. Le modèle d'architecture AAA (rectangle 2 de la figure 3.1) permet de décrire un grand nombre de machines. En effet ce modèle utilise un niveau d'abstraction qui n'est pas trop élevé de façon à ne pas négliger des détails importants de l'architecture, mais aussi, qui n'est pas

trop faible rendant le temps que prend le processus d'optimisation très long (c'est un niveau macro-RTL). Ce modèle se base sur un graphe (S, A) où S est l'ensemble des sommets et A l'ensemble des arcs bidirectionnels les reliant. Chaque élément de S représente une machine à états finis qui peut appartenir à un seul de ces quatre ensembles [Grandpierre, 2000] :

* S_{OPR} : l'ensemble des opérateurs,
* S_M : l'ensemble des mémoires,
* S_{Bus} : l'ensemble des Bus/Mux/Demux avec ou sans arbitre,
* S_{Com} : l'ensemble des communicateurs.

Les opérateurs

Chaque opérateur représente une ou plusieurs unitées de traitement, le séquenceur qui les commandes et éventuellement les unités d'entrées-sorties connectées aux capteurs et actionneurs. Si une opération regroupe un ensemble d'instructions, les opérateurs peuvent être assimilés aux séquenceurs d'instructions des machines SIMD et MIMD. Chaque opérateur exécute séquentiellement un sous-ensemble des opérations du graphe d'algorithme sur des données stockées dans les registres qui lui sont connectés. Les opérations peuvent être de calcul ou d'entrée-sortie. Une opération de calcul consomme, à chaque itération, des données dans un ou plusieurs registres connectés à l'opérateur. Elle modifies ces données pour produire de nouvelles données qu'elle écrit dans un ou plusieurs registres. Les opérations d'entrée produisent des données à partir d'informations physiques grâce aux capteurs inclus dans l'opérateur, alors que les opérations de sorties transforment les données en grandeurs physiques grâce aux actionneurs.

Un opérateur est caractérisé par :

- La liste d'opérations qu'il est capable d'exécuter et pour chacune de ces opérations sa durée d'exécution qui est notée $\delta(O, P)$
- Le nombre maximal d'arcs pouvant y être connectés et pour chacun de ces arcs sa bande passante maximale.

Les mémoires

Un registre permet de sauvegarder une donnée par le mécanisme d'écriture et d'y accéder par le mécanisme de lecture effectué par un opérateur (ou un communicateur) qui lui est connecté.Une mémoire est constituée de plusieurs registres identiques ainsi que de bus, multiplexeur, démultiplexeur permettant l'accès à ces registres.

Les mémoires peuvent être de deux types : "Random Acces Memory" (RAM) ou "Sequenciel Acces Memory" (SAM). L'accès aux registres qui forment une RAM est aléatoire, c'est-à-dire qu'il n'impose pas d'ordre entre lecture et écriture. Chaque mémoire RAM est caractérisée par sa taille et sa bande passante BP_{max} qui est égale à la bande passante des registres qui la constituent. La différence des bandes passantes permet de modéliser la hiérarchie mémoire connectée à un opérateur. Une mémoire RAM peut être utilisée pour enregistrer les instructions du programme, on l'appelle alors RAM_P (mémoire programme). Une mémoire RAM peut ne contenir que les données à traiter, elle est appelée alors RAM_D (mémoire données). Les mémoires utilisées pour enregistrer à la fois les instructions et les données sont appelées RAM_{DP}. Une mémoire RAM peut être soit

connectée à un seul opérateur soit partagée entre plusieurs opérateurs et communicateurs. Dans ce dernier cas, elle peut être utilisée pour la communication.

Les mémoires SAM, à l'inverse des RAM, imposent que les messages qu'elles contiennent soient lus dans le même ordre de leurs écritures. Les mémoires SAM sont essentiellement utilisées pour le passage de messages de communication entre les différents séquenceurs (processeurs et communicateurs). Une SAM peut modéliser un buffer FIFO (First In First Out, premier entré premier sorti en français) point à point ou multipoint.

Les mémoires sont caractérisés par :

- Leurs tailles.
- Leurs bandes passantes maximales BP_{max}.

Les bus

Un sommet bus modélise un bus, un multiplexeur et un démultiplexeur, un décodeur d'adresse et éventuellement un arbitre pour gérer l'accès aux ressources partagées. Il permet de modéliser l'accès d'un opérateur à un parmi n registres qui lui sont connectés par le même arc ou aux registres partagés entre cet opérateur et un autre.

Les communicateurs

Un communicateur exécute séquentiellement des opérations de communication. Une opération de communication est un ensemble d'instructions de transfert de données entre deux mémoires qui sont connectées au communicateur qui l'exécute. Pour pouvoir exécuter ces transferts, un communicateur doit être paramétré par un opérateur. Il peut être utilisé pour modéliser un canal de DMA [Dou et al., 2008].

Chaque communicateur est caractérisé par :

- La durée de son paramétrage.
- La durée de transfert pour chaque type de données.

Exemples de graphes d'architecture

L'architecture la plus simple est l'architecture mono-processeur. Cette architecture comporte un séquenceur d'instructions et son unité arithmétique et logique (UAL) connectée à une ou plusieurs mémoires. La figure 3.4 représente le modèle d'une architecture mono-processeur comportant un opérateur ("Opr"), modélisant l'unité arithmétique et logique, connecté à deux mémoires : une mémoire externe ("RAM E") et une autre interne ("RAM I"). Le cadre en pointillé sur la figure 3.4 représente la limite de la puce contenant le processeur. Pour différencier les mémoires externes des mémoires internes, on associe aux mémoires internes des bandes passantes plus élevées que celles attribuées aux mémoires externes [Grandpierre, 2000], ceci est effectué en associant à l'arc reliant l'opérateur à la mémoire interne une bande passante plus élevée comme expliqué plus haut.

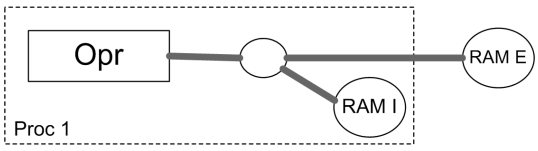

FIGURE 3.4 – Modèle d'une architecture mono-processeur à deux mémoires

Il est de plus en plus fréquent que les processeurs soient munis d'un DMA pour effectuer le transfert de données entre les mémoires externes et internes [Dou et al., 2008]. On peut modéliser les architectures mono-processeurs munis d'un DMA par le modèle représenté par la figure 3.5. Ce modèle comporte, en plus de l'opérateur et les mémoires, un communicateur ("Com"). Le communicateur connecté aux deux mémoires modélise le canal de DMA effectuant le transfert de données entre ces deux mémoires : "RAM E" (externe) et "RAM I" (interne).

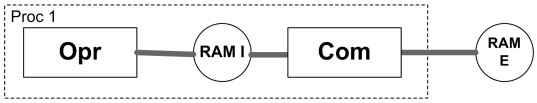

FIGURE 3.5 – Modèle d'une architecture mono-processeur comportant un DMA

La figure 3.6 représente le modèle d'une architecture bi-processeur. Chaque processeur comporte une UAL, une mémoire RAM ("R1" et "R2") et un DMA ("Com 1" et "Com 2"). Le transfert de données inter-processeur se fait par les DMA à travers une mémoire FIFO. La mémoire FIFO est modélisée par une mémoire SAM ("S").

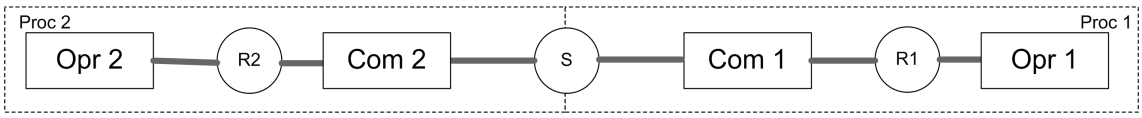

FIGURE 3.6 – Modèle d'une architecture biprocesseur

La figure 3.7 représente un modèle d'architecture comportant quatre processeurs connectés en anneau. Ces processeurs sont connectés entre eux à travers des FIFOs modélisés par des mémoires SAM ("S1", "S2", "S3" et "S4").

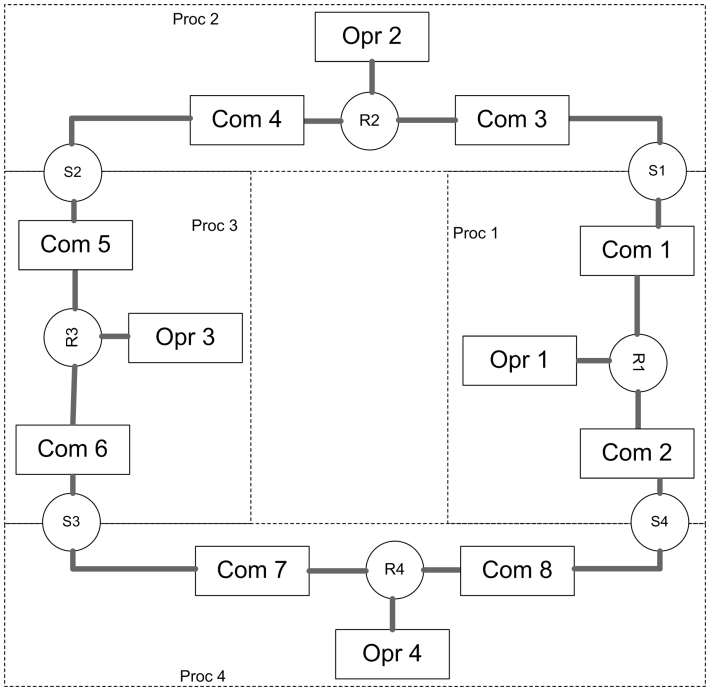

FIGURE 3.7 – Modèle d'architecture à quatre processeurs en anneau

La figure 3.8 représente une architecture à quatre processeurs connectés entre eux à travers une mémoire partagée. Chaque processeur dispose d'un communicateur (Com) qui permet d'accéder à la mémoire RAM partagée (SR).

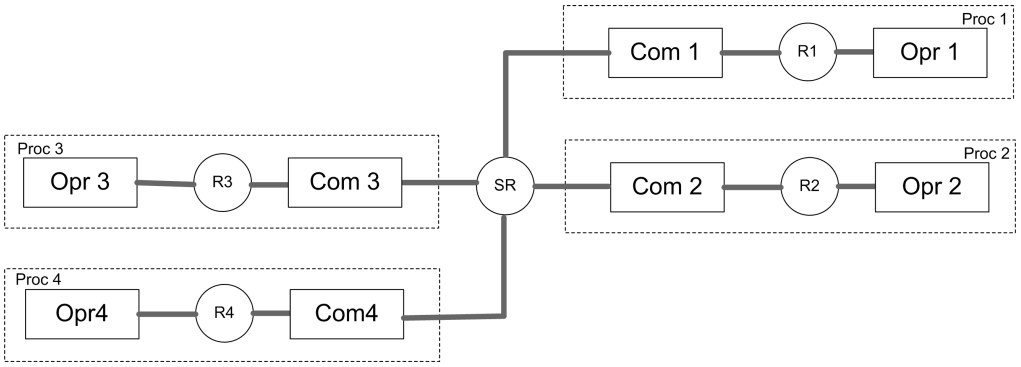

FIGURE 3.8 – Modèle d'architecture à quatre processeurs connectés à travers une mémoire partagée

3.3.3 Modèle d'implantation

L'implantation est modélisée par un graphe d'implantation obtenu en transformant le graphe d'algorithme en tenant compte du graphe d'architecture. Un exemple de graphe d'implantation d'un algorithme contenant quatre opérations sur une architecture biprocesseur est donné par la figure 3.9. Pour obtenir le graphe d'implantation, les opérations du graphe d'algorithme sont distribuées puis ordonnancées sur les opérateurs du graphe d'architecture. Nous verrons en section 3.3.5 qu'une heuristique est utilisée pour réaliser cette distribution ordonnancement.

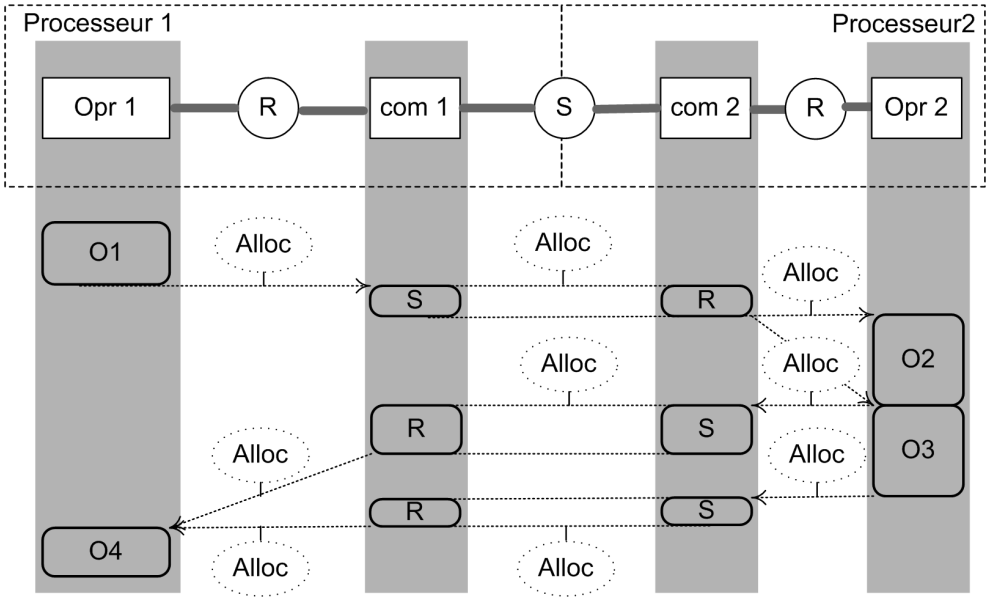

FIGURE 3.9 – Exemple de graphe d'implantation

Distribution

Lors de la distribution, une allocation spatiale du graphe d'algorithme sur le graphe d'architecture est effectuée. Cette opération se produit en trois étapes : le routage, le partitionnement et la communication.

* Routage : le routage consiste à construire le plus court chemin entre chaque couple d'opérateurs et de communicateurs du graphe d'architecture.
* Partitionnement : il consiste à découper le graphe d'algorithme en plusieurs éléments de partition. Chaque sous-graphe sera exécuté par l'opérateur du graphe d'architecture sur lequel il a été alloué.
* Communication : après partitionnement, on dit qu'il y a une dépendance de données inter-processeurs lorsque l'opération productrice et l'opération consommatrice d'une donnée sont distribuées sur des opérateurs différents. Ces dépendances de données inter-processeur nécessitent un transfert de données de l'opérateur exécutant l'opération productrice à l'opérateur exécutant l'opération consommatrice en passant par un chemin (trouvé lors du routage) reliant ces deux opérateurs. La communication, consiste à ajouter un nouveau sommet allocation sur chaque mémoire ou bus d'un chemin du graphe d'architecture reliant les deux opérateurs producteur et consommateur de la donnée à transférer.

Ordonnancement

Vu le caractère séquentiel des opérateurs de calcul et de communication, il faut définir pour chacun de ces opérateurs un ordre total d'exécution des opérations qu'il exécute. Cet ordre doit respecter l'ordre partiel imposé par les dépendances de données du graphe d'algorithme ainsi que la disponibilité de ressources matérielles des opérateurs et des communicateurs.

3.3.4 Génération automatique des exécutifs

La méthodologie AAA vise une implémentation optimisée de l'algorithme sur l'architecture cible, mais aussi, la génération automatique d'un fichier d'exécutifs pour chaque opérateur de cette architecture (rectangle 4 de la figure 3.1).

Dans un premier temps, le graphe d'implémentation est traduit en un macro-codes génériques, c'est à dire indépendant du langage de programmation de l'opérateur cible. Chacun de ces macros est formé d'une partie d'allocation de mémoire, une partie de séquences de communications et une autre partie de séquence d'opérations de calcul. Ensuite, une bibliothèque spécifique à chaque langage de programmation est utilisée pour traduire ce macro code dans le langage du processeur cible.

Ce processus de génération d'exécutifs scindé en deux étapes, une indépendante de l'architecture cible et l'autre dépendante, peut ainsi s'adapter à différents types de langages de programmation, donc à plusieurs types d'architectures. Plus de détails concernant la génération automatique des exécutifs seront donnés dans la section 5.2.

3.3.5 Optimisation

La méthodologie AAA vise à réduire la latence d'exécution de l'algorithme. Pour pouvoir estimer la latence, il faut calculer les dates de début et de fin associées aux

graphes d'algorithme et d'implantation. Ces dates seront utilisées pour le calcul de la fonction coût de l'heuristique d'optimisation.

Calcul des dates de début et de fin d'exécution des opérations

Comme vu en 3.3.1, chaque opération est caractérisée par sa durée d'exécution sur chaque opérateur capable de l'exécuter. Avant distribution, on ne connait pas l'opérateur qui va exécuter l'opération O_i du graphe d'algorithme. Donc on ne peut pas connaitre sa durée d'exécution. Cette durée est approximée par la moyenne Δ_{app} des durées d'exécution de l'opération O_i sur tous les opérateurs pouvant l'exécuter :

$$\Delta_{app}(O_i) = \frac{1}{n}\sum_{j=1}^{n}\delta(O_i, P_j) \tag{3.1}$$

où n est le nombre d'opérateurs capables d'exécuter l'opération O_i

La date de début au plus tôt depuis le début $S(O_i)$ est définie par :

$$S(O_i) = \begin{cases} 0 & \text{si } \Gamma^{-1}(O_i) = \emptyset \\ \max_{\forall O_j \in \Gamma^{-1}(O_i)} E(O_j) & \text{sinon} \end{cases} \tag{3.2}$$

Avec $\Gamma^{-1}(O_i)$ l'ensemble des prédécesseurs de O_i.

On définit aussi la date de fin au plus tôt définie depuis le début $E(O_i)$ par la somme de la date de début au plus tôt depuis le début $S(O_i)$ et de la durée d'exécution approximée $\Delta_{app}(O_i)$:

$$E(O_i) = S(O_i) + \Delta_{app}(O_i) \tag{3.3}$$

On définit aussi les dates au plus tard depuis la fin. Ceci sera utile pour ne pas être amené à manipuler des dates négatives. La date de fin au plus tard depuis la fin $\bar{E}(O_i)$ d'une opération O_i correspond à la plus grande date de début au plus tard depuis la fin $\bar{S}$ de ses successeurs si O_i possède des successeurs, si non, cette date est nulle.

$$\bar{E}(O_i) = \begin{cases} 0 & \text{si } \Gamma(O_i) = \emptyset \\ \max_{\forall O_j \in \Gamma(O_i)} \bar{S}(O_j) & \text{sinon} \end{cases} \tag{3.4}$$

Avec $\Gamma(O_i)$ est l'ensemble des successeurs de O_i.

La date de début au plus tard depuis la fin $\bar{S}(O_i)$ d'une opération O_i est la somme de la date de fin au plus tard depuis la fin de O_i et de sa durée approximée d'exécution $\Delta_{app}(O_i)$.

$$\bar{S}(O_i) = \bar{E}(O_i) + \Delta_{app}(O_i) \tag{3.5}$$

Les dates que nous venons de présenter ne tiennent compte que du graphe d'algorithme. Elles sont pour le moment indépendantes de l'implantation. Lors de l'implantation, chaque opération du graphe d'algorithme est associée à un opérateur qui va l'exécuter. Donc on ne parle plus de durée d'exécution approximée mais de durée d'exécution de l'opération O_i sur l'opérateur P notée $\delta(O_i, P)$. Soit la relation Π qui associe à chaque opération O_i du graphe d'algorithme l'opérateur P sur lequel elle est distribuée $\Pi(O_i) = P$. La date de début au plus tôt définie depuis le début $S(O_i)$ tient compte désormais de la date de fin de la dernière opération exécutée sur l'opérateur $\Pi(O_i)$.

$$S(O_i) = \begin{cases} 0 & \text{si } \Gamma^{-1}(O_i) = \Gamma'^{-1}(O_i) = \emptyset \\ \max(\max_{\forall O_j \in \Gamma^{-1}(O_i)} E(O_j), E(\Gamma'^{-1}(O_i))) & \text{sinon} \end{cases} \quad (3.6)$$

Avec $\Gamma'^{-1}(O_i)$ la relation qui associe à chaque opération O_i l'ensemble de ses prédécesseurs distribués sur le même opérateur.

La date de fin au plus tôt définie depuis le début devient :

$$E(O_i) = S(O_i) + \delta(O_i, \Pi(O_i)) \quad (3.7)$$

De la même façon, la date de fin au plut tard et de début au plus tard définies depuis la fin deviennent :

$$\bar{E}(O_i) = \begin{cases} 0 & \text{si } \Gamma(O_i) = \emptyset \text{ et } \Gamma'(O_i) = \emptyset \\ \max(\max_{\forall O_j \in \Gamma(O_i)} \bar{S}(O_j), \bar{S}(\Gamma'(O_i))) & \text{sinon} \end{cases} \quad (3.8)$$

$$\bar{S}(O_i) = \bar{E}(O_i) + \delta(O_i, \Pi(O_i)) \quad (3.9)$$

Avec $\Gamma'(O_i)$ la relation qui associe à chaque opération O_i l'ensemble de ses successeurs distribués sur le même opérateur.

heuristique d'optimisation

Pour un algorithme d'application de traitement de signal réel et une architecture multi-composants donnés, il existe un nombre très élevé de solutions possibles. Le parcours de toutes ces solutions possibles pour en extraire la solution optimale nécessite un temps tellement grand qu'il est irréaliste d'envisager de trouver cette solution. Face à ce problème NP difficile, les méthodologies de prototypage rapide utilisent des heuristiques pour réduire l'ensemble de solutions à explorer.

La méthodologie AAA utilise une heuristique gloutonne basée sur une méthode de liste pour aboutir à une implémentation optimisée d'un algorithme sur une architecture donnée. L'intérêt de ce choix a été défendu dans [Vicard and Sorel, 1998]. Cette heuristique vise à minimiser la longueur du chemin critique (figure 3.10) qui est le chemin le plus long du graphe d'algorithme.

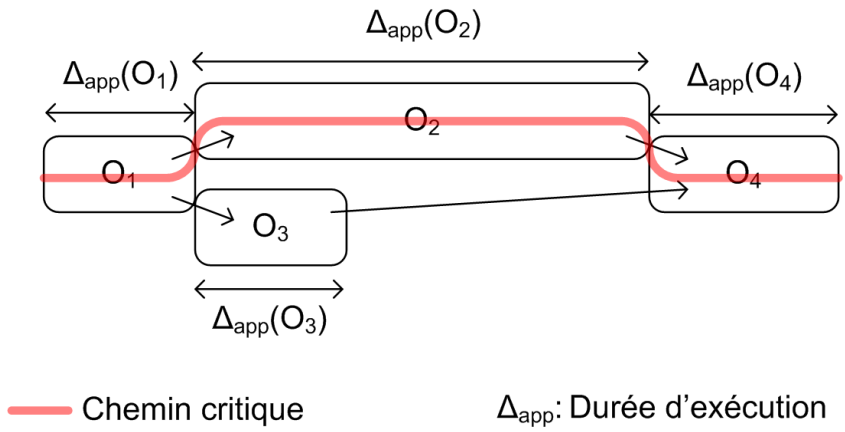

FIGURE 3.10 – Représentation du chemin critique

La fonction coût σ de l'heuristique AAA, appelée pression d'ordonnancement, tient compte de l'éventuel allongement du chemin critique à la suite de l'ordonnancement de l'opération O_i et de la flexibilité d'ordonnancement de cette opération O_i. La flexibilité d'ordonnancement $F(O_i)$ (figure 3.11) est la différence entre la plus grande date de début de l'opération O_i n'engendrant pas de rallongement du chemin critique et sa plus petite date de début possible. Cette pression d'ordonnancement est donnée par la différence entre la pénalité d'ordonnancement $P(O_i, P)$ et la flexibilité d'ordonnancement $F(O_i, P)$ ($\sigma(O_i, P) = P(O_i, P) - F(O_i, P)$).

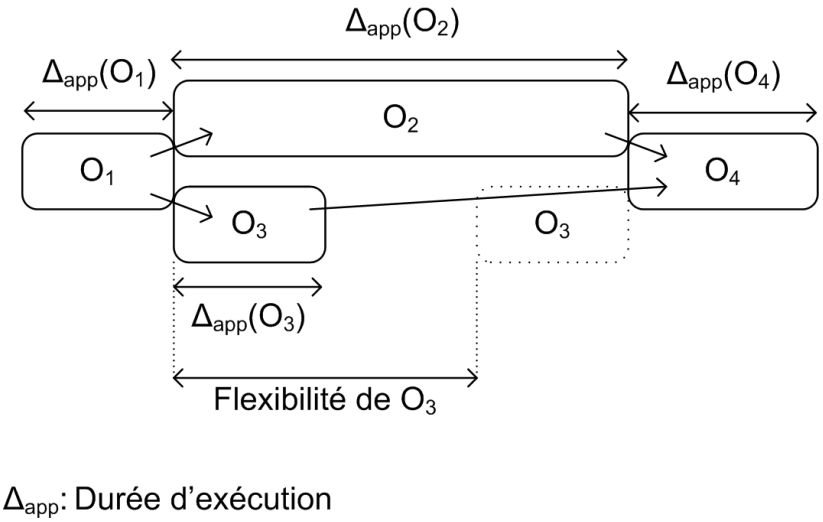

FIGURE 3.11 – Représentation de la flexibilité d'une opération

L'heuristique AAA est présentée dans l'algorithme 1. Elle commence par l'initialisation de la liste L en y mettant les opérations qui n'ont pas de prédécesseurs. Ensuite, tant qu'il reste encore des opérations à ordonnancer dans la liste, elle cherche tour à tour pour chacune d'elles l'opérateur permettant d'obtenir la plus petite date de fin au plus tôt. Puis, parmi la liste d'opérations, elle cherche celle qui maximise la fonction coût σ pour l'ordonnancer sur l'opérateur qui lui a été associé dans l'étape précédente. Lors de l'ordonnancement des opérations de calculs, les communications nécessaires sont prises en compte. Donc des sommets allocations sont ajoutés automatiquement sur chaque mémoire ou bus utilisé pour établir ces communications. Enfin, la liste est mise à jour en enlevant l'opération qui vient d'être ordonnancée et en y ajoutant ses succésseurs qui deviennent ordonnançables (dont tous les prédécesseurs sont ordonnancés).

Algorithm 1 Heuristique AAA pour les composants programmables

Require: Graphe d'algorithme G_{AL} et graphe d'architecture G_{AR}
Ensure: Graphe d'implémentation optimisé
1: $L = \{O_i \in G_{AL} / \Gamma^{-1}(O_i) = \emptyset\}$
2: **while** $L \neq \emptyset$ **do**
3: **for all** $O_i \in L$ **do**
4: Chercher l'opérateur $P(O_i) \in G_{AR}$ permettant d'obtenir la plus petite date de fin au plus tôt
5: **end for**
6: Choisir $O_i \in L$ qui maximise la fonction coût σ et l'ordonnancer sur l'opérateur $P(O_i)$ trouvé en 4
7: Mettre à jour la liste L
8: **end while**

3.3.6 Outil logiciel : SynDEx

Le logiciel principal qui implante la méthodologie AAA s'appelle SynDEx (Synchronized Distributed Executive) [Lavarenne et al., 1991]. C'est un logiciel de prototypage rapide d'applications temps réel distribuées embarquées. Son interface principale permet de définir graphiquement le graphe d'algorithme, le graphe d'architecture et de lancer l'adéquation et la génération automatique d'exécutifs. Une capture écran de l'interface actuelle est donnée par la figure 3.12. La fenêtre de gauche représente un algorithme composé d'un sommet acquisition, une constante, deux fonctions et deux sommets résultats (actionneurs). Celle de droite représente l'architecture formée par deux processeurs connectés. L'utilisation de ce logiciel est libre pour une utilisation non commerciale. Les premières versions de SynDEx ont été écrites en « smalltalk » (jusqu'à la Version 4) puis en « $C++$ » et depuis la Version 6 en « Caml ». C'est un langage fonctionnel puissant particulièrement adapté à la manipulation des listes telle que celles utilisées par l'heuristique. Le graphe d'implémentation est affiché par SynDEx sous forme de diagramme temporel (figure 3.13) qui contient autant de colonnes que de références d'opérateurs et de moyens de communication dans l'architecture principale. Chaque colonne contient l'ensemble d'opérations que cet opérateur ou moyen de communication doit exécuter.

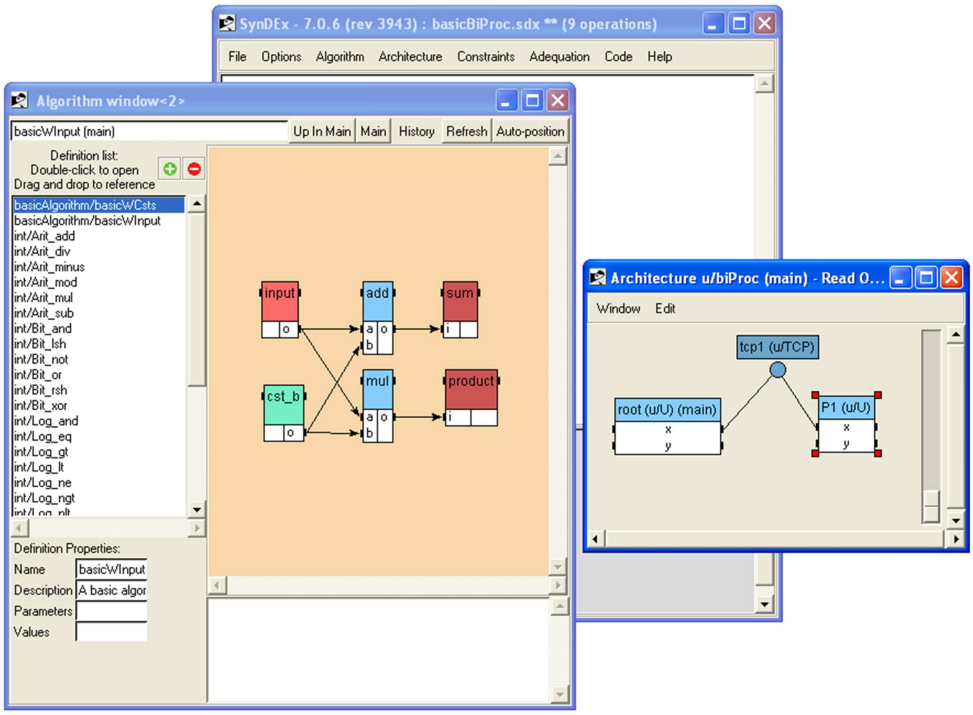

FIGURE 3.12 – Interface de l'outil SynDEx-7.0.6

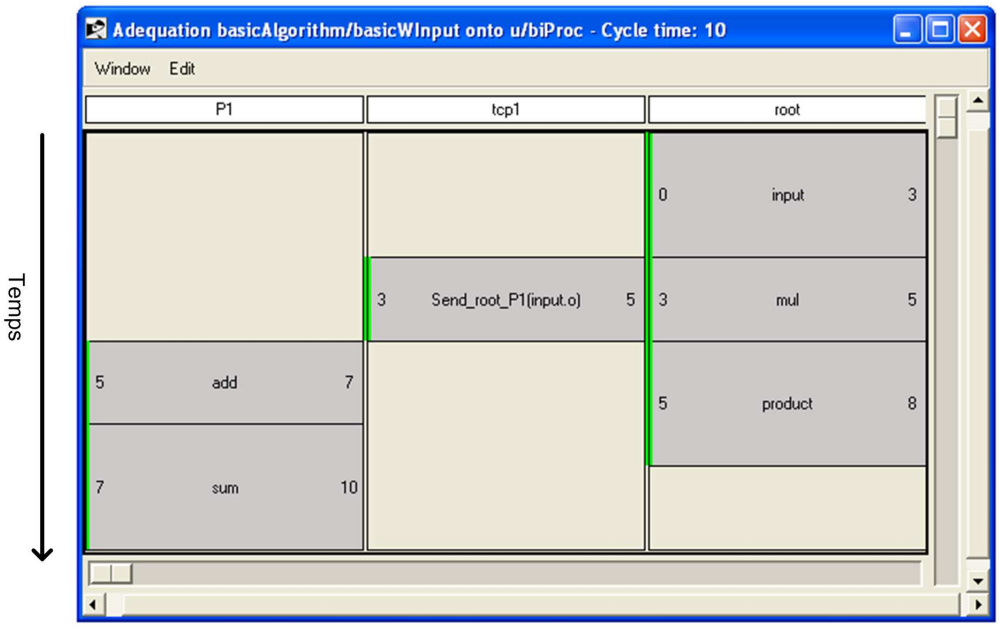

FIGURE 3.13 – Exemple de diagramme temporel

3.4 AAA pour les circuits reconfigurables : SynDEx-IC

La méthodologie AAA a été étendue pour supporter les composants reconfigurables et la synthèse de circuits [Kaouane, 2004, Niang et al., 2004]. Cette extension permet de générer une implémentation optimisée, c'est à dire une implémentation qui respecte une contrainte temporelle imposée ou s'en rapproche tout en minimisant la surface. Dans cette extension, l'optimisation ne consiste plus en une distribution et un ordonnancement puisque la cible est un FPGA ou un circuit unique. L'optimisation se fait par défactorisa-

tion partielle ou total du graphe d'algorithme. En effet l'implémentation d'une répétition qui ne contient pas d'inter dépendance entre les différentes itérations peut se faire de différentes manières : complètement séquentielle, complètement parallèle et toutes les implantations intermédiaires. La figure 3.14 illustre cet espace d'implantations possible. Nous verrons en détail ces différentes implantations page 55 à l'aide d'un exemple concret. Le graphe d'algorithme qu'utilise cette extension est le même que celui présenté précédemment. Par contre le graphe d'architecture subit des modifications pour respecter les spécificités des FPGA. Cette extension de AAA est implémentée dans un outil appelé SynDEx-IC.

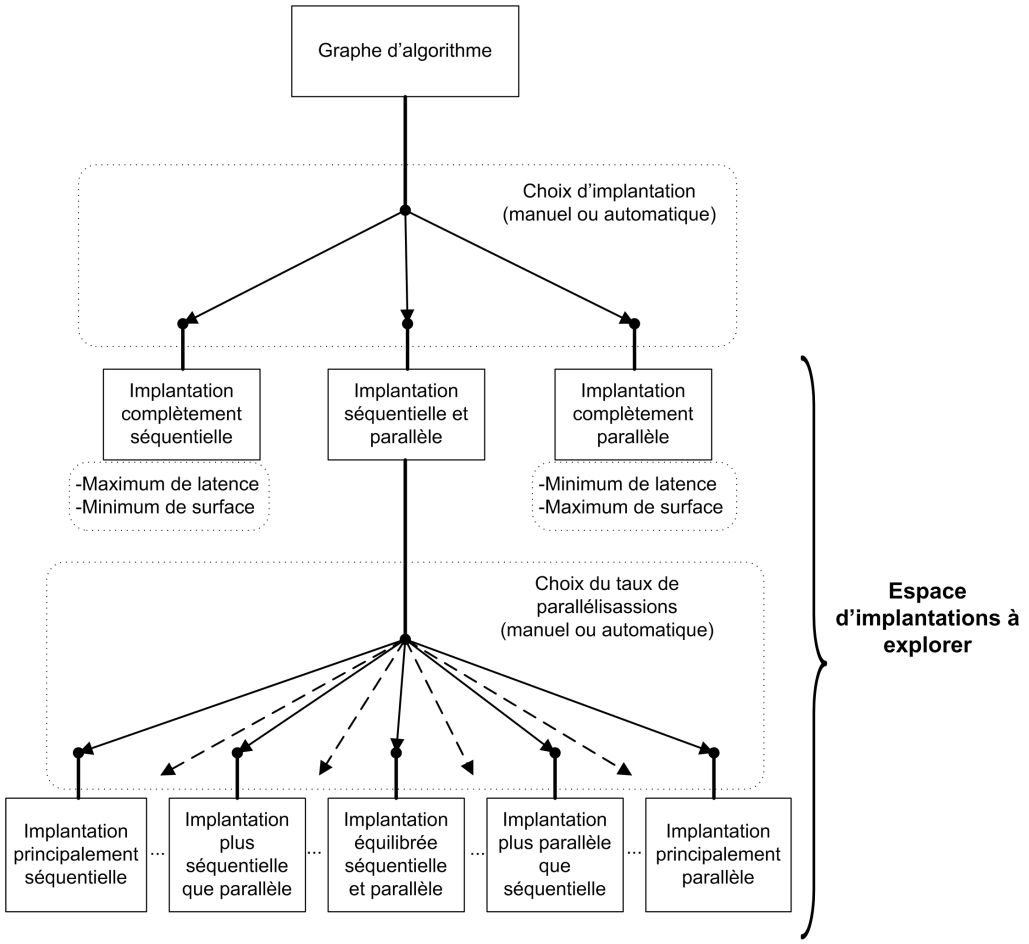

FIGURE 3.14 – Espace d'implantation matérielle

3.4.1 Modèle d'algorithme

Cette extension de la méthodologie AAA est basée sur le même modèle d'algorithme présenté au paragraphe 3.3.1. Cette extension exploite d'avantage la notion de factorisation. En effet, la méthodologie AAA pour les circuits programmables défactorise systématiquement toutes les frontières de factorisation avant de commencer l'opération de partitionnement/ordonnancement. Au contraire, l'extension de AAA pour les circuits reconfigurables tend à chercher une défactorisation optimisée de ces frontières de factorisation. Cette défactorisation optimisée satisfait la contrainte temporelle tout en utilisant le minimum de blocs logiques du FPGA.

3.4.2 Modèle d'architecture

Nous avons vu que le graphe d'architecture constitue une des entrées de la méthodologie AAA pour les composants programmables. Par contre, pour la méthodologie AAA pour les circuits reconfigurables, le graphe d'architecture est le circuit à générer, donc constitue un résultat. A partir du graphe d'algorithme factorisé, et après l'exploration de l'espace des solutions pour trouver une implantation de l'algorithme qui satisfait les contraintes temporelles, on obtient finalement un graphe d'architecture par traduction directe du graphe d'algorithme transformé. Chaque sommet de ce graphe représente un opérateur et chaque arc une connexion inter opérateurs. Ce graphe contient aussi une partie contrôle composée de sommets contrôle interconnectés. Nous allons par la suite spécifier les transformations qui permettent d'obtenir le chemin de données et de contrôle à partir du graphe d'algorithme.

Synthèse du chemin de données

La synthèse du chemin de données consiste à remplacer chaque sommet du graphe d'algorithme par un opérateur et chaque dépendance de données par une connexion interconnectant l'opérateur producteur aux opérateurs consommateurs de cette donnée. A chaque type de sommet du graphe d'algorithme correspond un type d'opérateur. Nous allons ci-dessous présenter les principaux types des sommets du graphe d'algorithme, représentés par des cercles, et les opérateurs correspondants, représentés par des rectangles.

* Opérateur calcul : chaque sommet opération du graphe d'algorithme est remplacé par un circuit réalisant cette fonction. Ce circuit est appelé opérateur calcul. Il peut être arithmétique ou logique, combinatoire ou séquentiel. La figure 3.15 représente un exemple de sommet opération du graphe d'algorithme et l'opérateur calcul correspondant. Cet opérateur calcul peut être pris d'une bibliothéque VHDL ou défini par l'utilisateur.

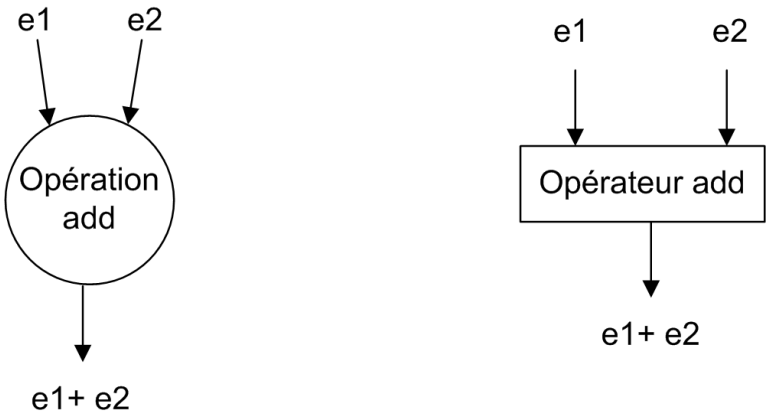

FIGURE 3.15 – Sommet opération additionneur du graphe d'algorithme et l'opérateur correspondant

* Opérateur implode : un opérateur implode, identifié par M, est utilisé pour effectuer un regroupement ordonné de d bus unidirectionnels en un seul. Cet opérateur permet de réaliser les sommets Join (présenté page 38) du graphe d'algorithme factorisé. Il peut être de deux types : parallèle ou séquentiel. Le choix entre les

deux sera effectué lors de la phase d'optimisation présentée page 55. La figure 3.16 représente le sommet Join ainsi que les opérateurs correspondants.

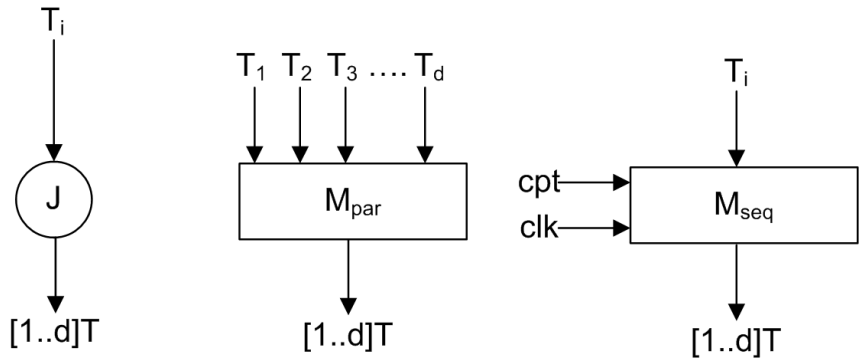

FIGURE 3.16 – Sommet implode du graphe d'algorithme et l'opérateur correspondant

L'opérateur Implode séquentiel (M_{seq}) permet d'implémenter le sommet Join du graphe d'algorithme en une répétition temporelle. Les signaux de contrôles qu'il reçoit sont le signal d'horloge (clk) et le compteur d'itération (cpt).

L'opérateur Implode parallèle (M_{par}) permet d'implanter le sommet Join du graphe d'algorithme en une répétition spatiale.

* Opérateur explode : cet opérateur réalise l'opération inverse de celle réalisée par l'opérateur implode. Cet opérateur, identifié par X, décompose un bus unidirectionnel en d sous-bus. Il est utilisé dans l'implémentation des sommets Fork (présenté page 38) du graphe d'algorithme. Il peut être, comme le montre la figure 3.17, de deux types : parallèle ou séquentiel.

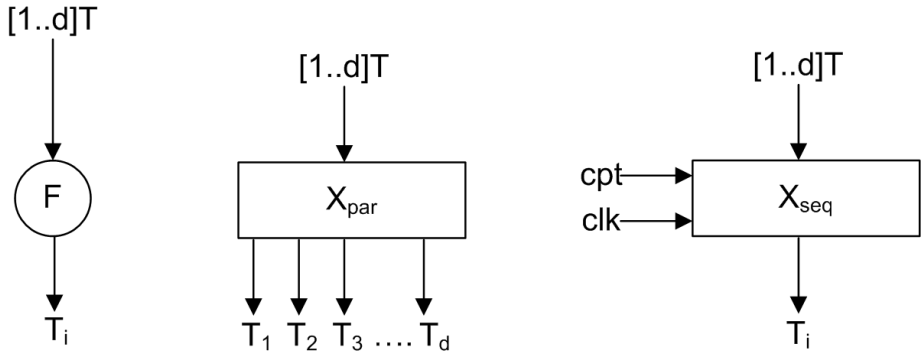

FIGURE 3.17 – Sommet explode du graphe d'algorithme et l'opérateur correspondant

L'opérateur explode séquentiel (X_{seq}) permet l'implantation du Fork en une répétition temporelle. Il reçoit en entrée les données traversant la frontière, le signal d'horloge et un compteur (cpt) pour compter le nombre d'itérations.

L'opérateur explode parallel (X_{par}) permet d'implanter le sommet Fork du graphe d'algorithme en une répétition spatiale.

* L'opérateur retard : cet opérateur est un registre qui produit en sa sortie la valeur qui était sur son entrée lors d'un cycle d'horloge précédent. Il permet de réaliser les sommets Delay du graphe d'algorithme (présenté page 37). Comme le montre la figure 3.18, cet opérateur reçoit une valeur d'initialisation (init) qui sera à la sortie

au premier cycle, la valeur de l'entrée actuelle (ei) et deux signaux de contrôle : le signal d'horloge (clk) et un signal pour déclencher le chargement (en).

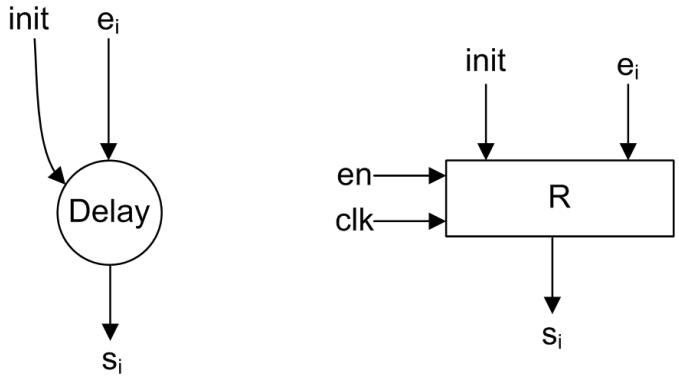

FIGURE 3.18 – Sommet Delay du graphe d'algorithme et l'opérateur correspondant

* L'opérateur Iterate : cet opérateur permet d'implanter les itérations inter motifs d'une factorisation. Il est identifié par I (figure 3.19) et reçoit le signal d'horloge (clk) en plus d'un compteur (cpt) comme signaux de contrôle.

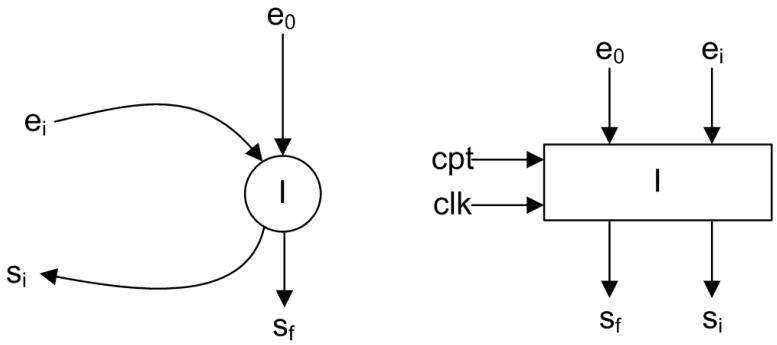

FIGURE 3.19 – Sommet iterate du graphe d'algorithme et l'opérateur correspondant

* L'opérateur Diffusion : cet opérateur, identifié par D (figure 3.20), permet d'implanter la diffusion matérielle. Il implante donc une connexion directe entre son entrée et sa sortie.

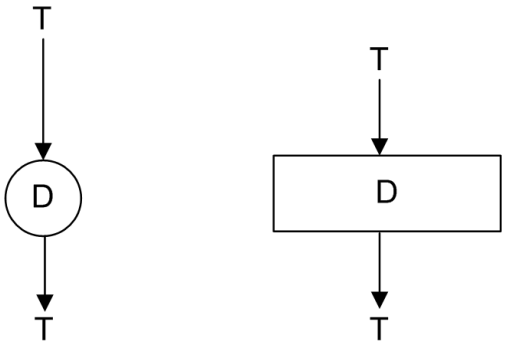

FIGURE 3.20 – Sommet Diffusion du graphe d'algorithme et l'opérateur correspondant

* L'opérateur Capteur : il est identifié par F^{∞}, il permet d'implanter le sommet Fork de la frontière infinie. Cet opérateur permet, comme le montre la figure 3.21, d'avoir le signal d'entrée à partir d'un signal physique. Un opérateur F^{∞} doit contenir tout les circuits permettant l'acquisition et la conversion du signal physique en signal numérique.

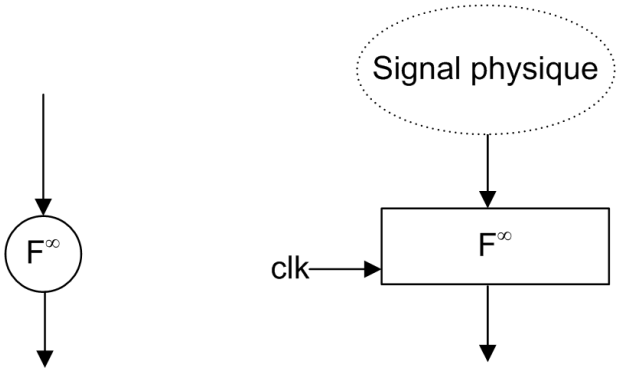

FIGURE 3.21 – Sommet Fork infinie du graphe d'algorithme et l'opérateur correspondant

* L'opérateur Actionneur : identifié par J^{∞}, il permet d'implémenter le sommet Join de la frontière infinie. Il permet de transformer le résultat de l'algorithme en signal physique. La figure 3.22 représente cet opérateur ainsi que le sommet correspondant.

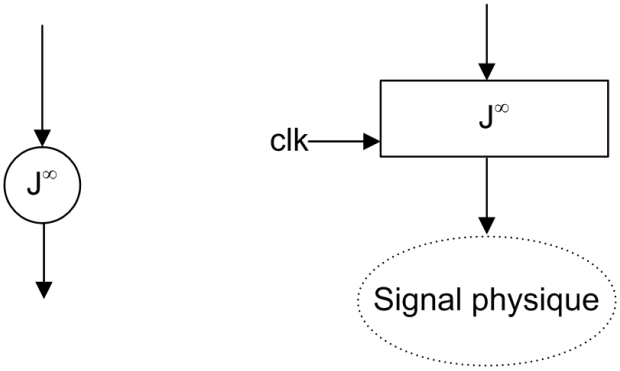

FIGURE 3.22 – Sommet Join infinie du graphe d'algorithme et l'opérateur correspondant

* L'opérateur Select : cet opérateur, représenté dans la figure 3.23, permet de choisir une sortie parmi plusieurs entrées suivant un ordre de priorité donné. Il permet d'implanter le conditionnement. Il doit intégrer les circuits permettant l'encodage de priorité et la sélection de la donnée.

Remarque : tous les opérateurs contenant des registres doivent avoir une entrée de contrôle de validation de chargement (en). Pour simplifier les figures, on ne les a pas représentés.

Synthèse du chemin de contrôle

Le chemin de contrôle correspond aux circuits nécessaires pour générer les signaux de contrôles pour le bon fonctionnement des multiplexeurs, registres et autres circuits du chemin de données. Il est responsable aussi de la synchronisation des opérations. Cette

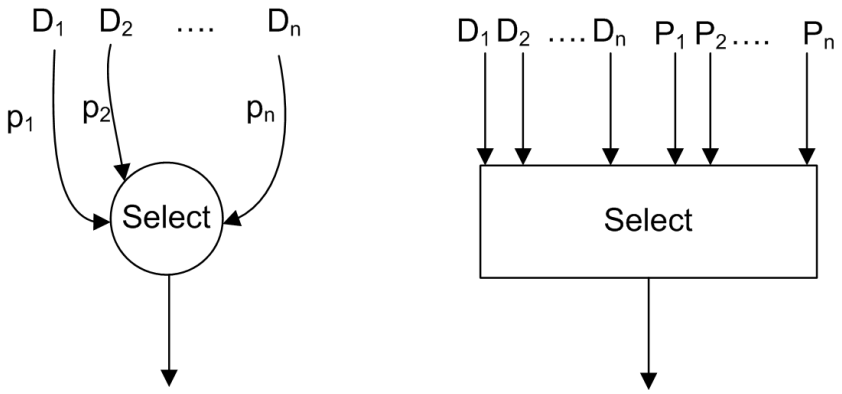

FIGURE 3.23 – Sommet select et l'opérateur correspondant

synchronisation est nécessaire pour garantir que le contenu d'un registre ne soit modifié que si les consommateurs de l'ancienne donnée aient déjà fini d'utiliser l'ancienne valeur et que la nouvelle valeur soit stable. La synchronisation est possible en utilisant un protocole de type requête/acquittement [Kaouane et al., 2004]. Lorsque les données nécessaires pour l'exécution des opérations d'une frontière F sont prêtes, l'unité de contrôle de cette frontière reçoit de chacune des unités de contrôles des frontières productrices de ces données un signal requête. Lorsque tous les signaux requêtes sont reçus, les opérations de cette frontière peuvent être exécutées. Après la consommation de ces données, l'unité de contrôle de F envoie un signal d'acquittement à chacun de ses prédécesseurs. C'est le même principe pour la relation de l'unité de contrôle de F et celles des frontières situées en aval de F.

En résumé, chaque unité de contrôle d'une frontière possède du côté amont (externe) deux ports pour la synchronisation : un port d'entrée requête indiquant la présence des données à consommer et un port de sortie acquittement indiquant aux frontières voisines que les données ont été utilisées. Du coté aval (interne), la synchronisation se fait aussi grâce à deux ports : un port de sortie requête indiquant aux frontières successeurs que les données dont ils ont besoin sont prêts et un port d'entrée acquittement indiquant que ces données sont consommées par les successeurs. La figure 3.24 montre un exemple d'unité de contrôle d'une frontière recevant deux données d1 et d2 de ses prédécesseurs et produisant une donnée d3 à son successeur. Lorsque la donnée d1 est prête, le signal r1 sera activé, de même pour le signal r2 indiquant la présence de la donnée d2. L'activation de ces deux signaux requête r1 et r2 active l'entrée requête de l'unité de contrôle de F indiquant que toutes les données à l'entrée de F sont prêtes. Si en plus le signal a3 est actif, les opérations de F peuvent commencer. Lorsque toutes les opérations de F sont terminées et la donnée d3 est prête, les signaux a1 et a2 sont activés pour indiquer que les données d1 et d2 sont consommées et un autre cycle peut commencer. Le signal r3 est activé pour indiquer au successeur de F que la donnée d3 est prête.

3.4.3 Modèle d'implantation : graphe de voisinage

Chaque frontière de factorisation est en relation avec une ou plusieurs autres frontières de factorisation. En effet chacune des frontières de factorisation consomme des données produites par d'autres frontières de factorisation et/ou produit des données pour d'autres frontières de factorisation. Ces relations de production/consommation de données sont résumées dans un graphe reprenant les liaisons entre chaque frontière de factorisation et

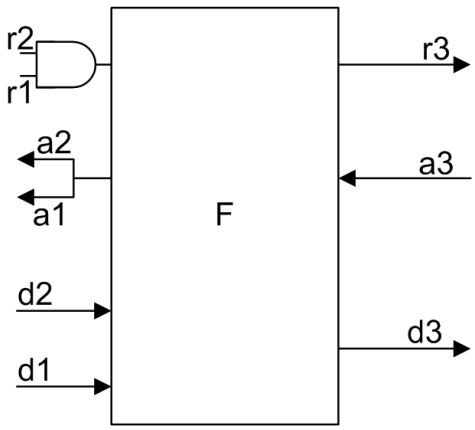

FIGURE 3.24 – Unité de contrôle de frontière synchronisée

ses voisins appelé graphe de voisinage. Chaque nœud de ce graphe de voisinage représente une frontière de factorisation et chaque arc représente un transfert de données d'une frontière productrice vers une autre consommatrice.

Chaque frontière de factorisation sépare une zone interne "rapide" (répétée) d'une externe "lente". En effet, chaque frontière de factorisation est rythmée par une horloge différente et ces côtés "lent" et "rapide" résultent de la fréquence d'échange de données au niveau des sommets de factorisation. Chaque frontière de factorisation peut aussi être consommatrice (en aval) ou/et productrice (en amant) par rapport à toute autre frontière voisine. Donc chaque sommet du graphe de voisinage peut être divisé en quatre régions : lent/amant, rapide/avale, rapide/amant et lent/aval. Ainsi chaque sommet du graphe de voisinage est défini par un quadruplet modélisant ces quatre régions.

Le graphe de voisinage est établi par transformation du graphe d'algorithme. Pour ce faire, on commence par identifier les dépendances internes à chaque frontière de factorisation. Ces dépendances internes ont pour source et destinataire deux sommets de factorisation d'une même frontière. Ainsi on crée les arcs du graphe de voisinage qui ont pour source et destination le même sommet. Puis pour chaque frontière, on cherche les dépendances externes reliant un sommet de factorisation de la frontière en question à un autre sommet de factorisation d'une autre frontière. Ainsi on crée les arcs reliant les différents sommets du graphe de voisinage.

Le graphe de voisinage établi permettra d'identifier les connexions entre les unités de contrôle correspondantes aux différentes frontières de factorisation [Kaouane et al., 2004].

3.4.4 Optimisation

Comme évoqué au début de la section 3.4, un FPGA peut exécuter un très grand nombre d'opérations en paralléle. Cette parallélisation permet de réduire le temps d'exécution de l'algorithme. En revanche, cela nécessite beaucoup de ressources donc induit une augmentation du nombre de blocs logiques du FPGA utilisé. Afin d'expliquer le processus d'optimisation, nous allons utiliser un exemple simple d'algorithme ne comprenant qu'une frontière de factorisation. Puis nous allons présenter les différentes implantations possibles pour un exemple simple. L'algorithme représenté par la figure 3.25 est composé de deux opérations d'acquisition "A" et "B", un sommet de calcul "C" et un sommet actuator "D". L'opération "C" est contenue dans une frontière de factorisation qui se répète quatre fois. En effet on remarque que "A" et "B" produisent quatre données par exécution et

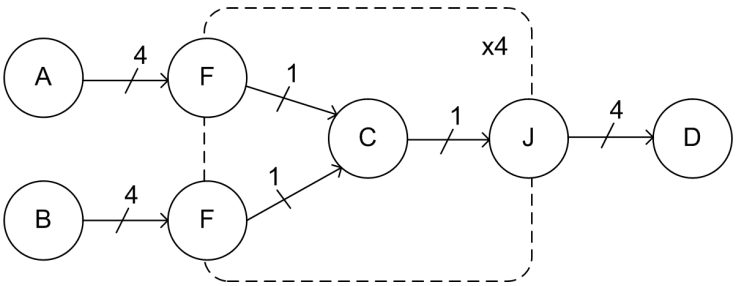

FIGURE 3.25 – Exemple de graphe d'algorithme

"D" en consomme aussi quatre. A partir de ce graphe d'algorithme, plusieurs implantations sont possibles allant de l'implantation totalement factorisée (synthèse de boucle) à l'implantation totalement défactorisée en passant par des implantations partiellement factorisées :

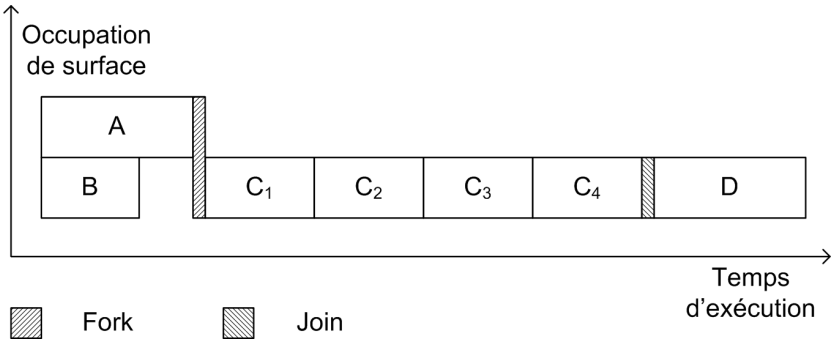

FIGURE 3.26 – Implantation complètement séquentielle

1. La figure 3.26 représente le diagramme temporel d'utilisation du FPGA pour une implantation complètement séquentielle. Dans cette implantation séquentielle, l'opération "C" a été transformée en un seul exemplaire de l'opérateur "C". Cet opérateur est utilisé quatre fois successives (séquentiellement) pour exécuter toutes les itérations de la frontière de factorisation. L'occupation de la surface du FPGA est minimale.

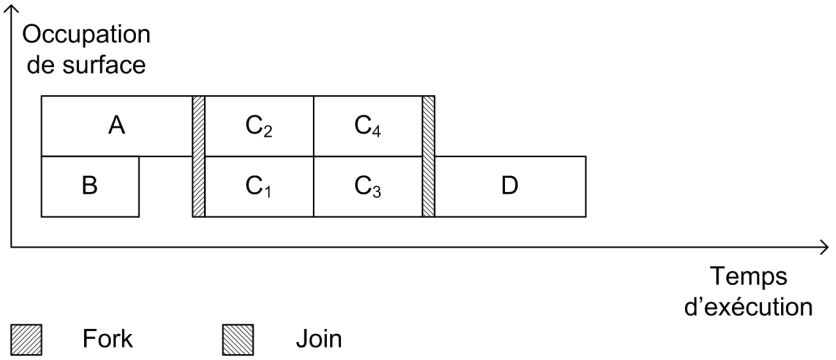

FIGURE 3.27 – Implantation partiellement factorisée

2. Le digramme de la figure 3.27 correspond à une implantation qui utilise deux exemplaires de l'opérateur "C". Chacun de ces deux exemplaires de l'opérateur "C" est utilisé deux fois pour pouvoir exécuter toutes les itérations de la frontière

de répétition. Cette implantation consomme plus de surface du FPGA mais réduit le temps d'exécution de l'algorithme.

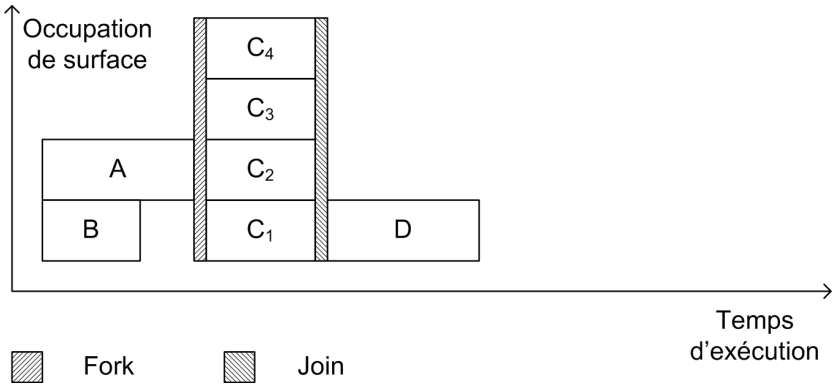

FIGURE 3.28 – Implantation complètement parallèle

3. Le dernier diagramme de la figure 3.28 correspond à l'utilisation de quatre exemplaires de l'opérateur "C". Cette dernière implantation est complètement parallèle : il suffit d'utiliser chaque opérateur "C" une fois pour exécuter toutes les itérations de la frontière de répétition. Cette implantation parallèle utilise une surface maximale du FPGA mais permet d'obtenir le temps minimal d'exécution de l'algorithme.

Donc **l'optimisation de l'implémentation d'un algorithme sur un FPGA revient à trouver une implémentation qui satisfait la contrainte temporelle tout en minimisant le nombre de blocs logiques utilisés**.

L'optimisation que fait la méthodologie AAA pour les circuits reconfigurables résulte de la défactorisation des frontières de factorisation (déroulage de boucles) que contient l'algorithme. La question qui se pose c'est : quelle frontière donne le meilleur compromis accélération/surcoût en surface lorsqu'elle est défactorisée et pour chaque frontière de factorisation choisie, quel est le facteur de défactorisation optimal ? Vu le grand nombre d'implémentations à explorer, il serait impossible d'explorer toutes les solutions pour des cas plus complexes, c'est un problème NP difficile. L'utilisation d'une heuristique s'impose pour réduire le temps de cette exploration des solutions possibles.

Une heuristique gloutonne et une autre heuristique de recuit simulé, ont été testées [Kaouane et al., 2004]. L'heuristique de recuit simulé donne de meilleurs résultats mais requière plusieurs heures de calcul. Dans l'extension de AAA que nous présentons dans cette thèse, nous verrons qu'une heuristique plus rapide est nécessaire, nous utiliserons donc l'heuristique gloutonne qui ne prend que quelques secondes ou minutes pour des cas complexes. Cette heuristique va défactoriser successivement certaines frontières et mesurer l'impact à l'aide de la fonction de coût donné par l'équation 3.10. Dans cette équation, ΔS représente l'augmentation du nombre de blocs logiques utilisés suite à la défactorisation de la frontière de factorisation. Le dénominateur $T - \max(T', C_t)$ correspond au gain temporel réalisé avec T, T' et C_t sont respectivement la latence de l'algorithme avant défactorisation, sa latence après défactorisation et la contrainte temporelle imposée.

$$f = \frac{\Delta S}{T - \max(T', C_t)} \tag{3.10}$$

Pour optimiser l'implémentation d'un algorithme, on utilise l'algorithme 2. Il commence par initialiser la liste de l'heuristique en y mettant les frontières de factorisation

se situant sur le chemin critique (ligne 2 de l'algorithme 2). Puis on associe à chacune des frontières de factorisation de la liste son facteur de défactorisation optimal qui est le minimum entre le plus grand facteur de défactorisation entrainant une diminution de la latence de l'algorithme et le plus petit permettant de satisfaire la contrainte temporelle imposée. Puis on calcule la fonction coût pour chaque couple (frontière de factorisation, facteur de défactorisation) et on en choisit celui qui minimise cette fonction. On réitère ces étapes jusqu'à avoir une latence inférieure ou égale à la contrainte temporelle imposée. Si cette contrainte temporelle ne peut pas être satisfaite, l'algorithme s'arrête lorsqu'on a le même résultat pour deux itérations successives. Dans ce cas, l'implémentation obtenue est complètement défactorisée. C'est l'implémentation qui se rapproche le plus de la contrainte temps imposée.

Algorithm 2 Heuristique AAA pour les composants reconfigurables

Require: Graphe d'algorithme G_{AL}
Ensure: Graphe d'implémentation optimisé
1: **while** $(T > C_t)$ ET $(T_1 \neq T)$ **do**
2: $T_1 = T$
3: $L = \{FF \in (G_{AL} \cap CR)\}$
4: **for all** $FF \in L$ **do**
5: associer à FF un facteur de défactorisation optimal k
6: **end for**
7: Défactoriser FF qui minimise f de son facteur de défactorisation optimal k
8: **end while**

3.4.5 Outil logiciel : SynDEx-IC

L'outil basé sur cette extension de AAA s'intitule SynDEx-IC [Niang et al., 2004]. Son interface (figure 3.29) repose sur celle celle de SynDEx v6. Elle permet de définir le graphes d'algorithme, de spécifier les caractéristiques du circuit et la contrainte temporelle, de lancer l'adéquation et de générer le code VHDL. Après avoir effectué l'adéquation, SynDEx-IC affiche le graphe de voisinage (figure 3.30) et les courbes de performances correspondant à l'implémentation choisie.

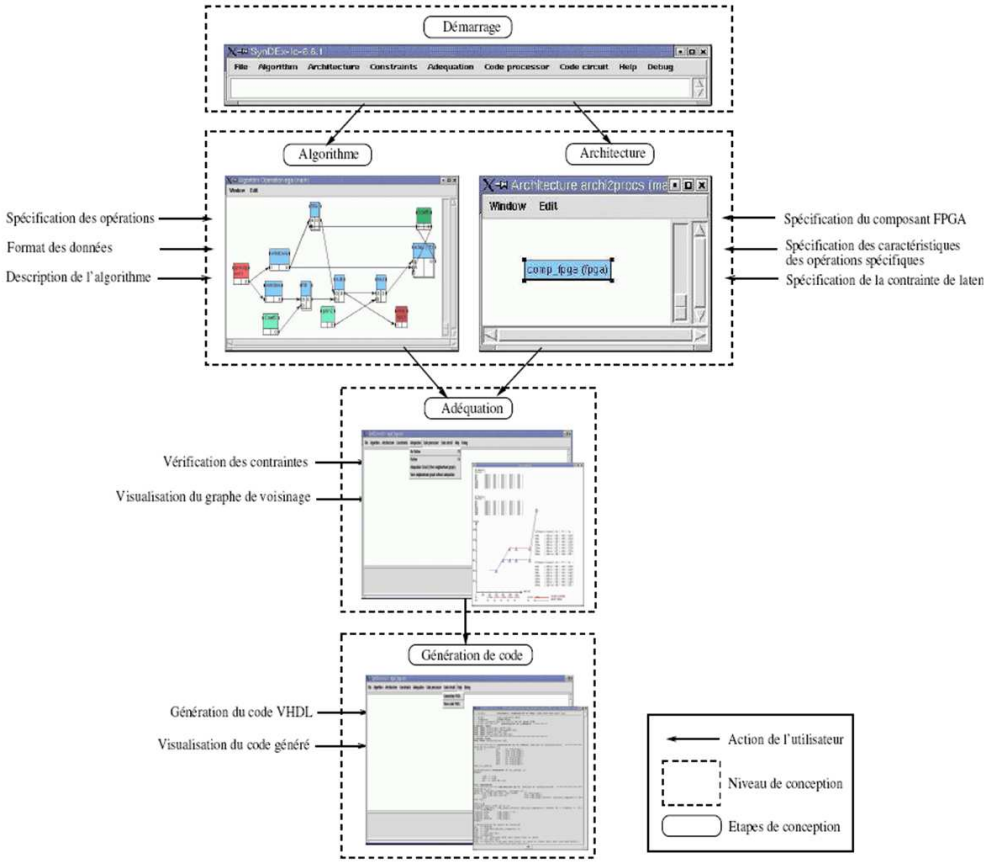

FIGURE 3.29 – Interface de l'outil SynDEx-IC

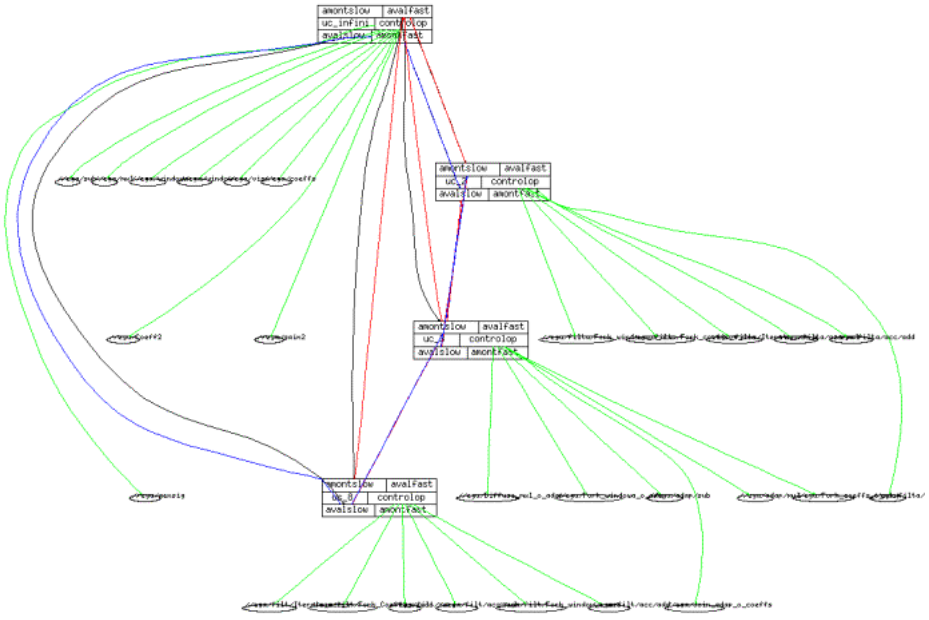

FIGURE 3.30 – Exemple de graphe de voisinage

3.5 Conclusion

Nous avons présenté dans ce chapitre la méthodologie AAA permettant d'optimiser l'implémentation des algorithmes sur des architectures multicomposants programmables.

Nous avons présenté aussi l'extension de cette méthodologie pour les circuits reconfigurables (mono-FPGA). Ces deux variantes de la méthodologie AAA s'adressent à 2 types d'architectures distinctes. Elles permettent le prototypage rapide des systèmes embarqués et permettent de réduire considérablement le temps de mise sur le marché de ces systèmes.

Mais la méthodologie AAA ne permet pas d'optimiser l'implémentation des algorithmes sur des architectures mixtes contenant des composants programmables et d'autres reconfigurables.

Le prochain chapitre a donc pour but d'étendre et coupler ces variantes et ces outils qu'il était donc nécessaire d'étudier en détail.

Chapitre 4

Extension des modèles AAA pour les composants reconfigurables et les architectures mixtes

4.1 Introduction

Comme nous l'avons vu dans le chapitre 3, le modèle d'architecture AAA ne permet pas de modéliser les architectures mixtes contenant à la fois des composants programmables et d'autres reconfigurables (FPGA). En effet, il ne permet pas de mettre en évidence le parallélisme offert par l'utilisation des FPGA. De plus, les outils implémentant aujourd'hui la méthodologie AAA ne traitent pas l'optimisation de l'implémentation des algorithmes sur des architectures mixtes. En effet, SynDEx traite le problème de partitionnement/ordonnancement des algorithmes sur des architectures multi-composants programmables. Il ne tient donc pas compte des spécificités de l'utilisation des FPGA. SynDEx-IC effectue l'optimisation de l'implémentation d'un algorithme sur une architecture mono FPGA pour satisfaire (ou s'approcher au plus de) une contrainte temporelle, donc il ne fait pas de partitionnement/ordonnancement et ne gére pas les communications entre les composants. Pour combler ce manque, nous nous proposons d'étendre les modèles utilisés aux circuit reconfigurables et mixtes. Ainsi, il sera possible d'utiliser la méthodologie AAA pour l'optimisation et la génération de codes des applications temps réel sur les architectures mixtes. La figure 4.1 montre le but que nous voulons atteindre par notre extension de la méthodologie AAA. En effet, nous visons de coupler les outils SynDEx et SynDEx-IC pour pouvoir optimiser l'implémentation des algorithmes sur les architectures mixtes et générer automatiquement les codes correspondants.

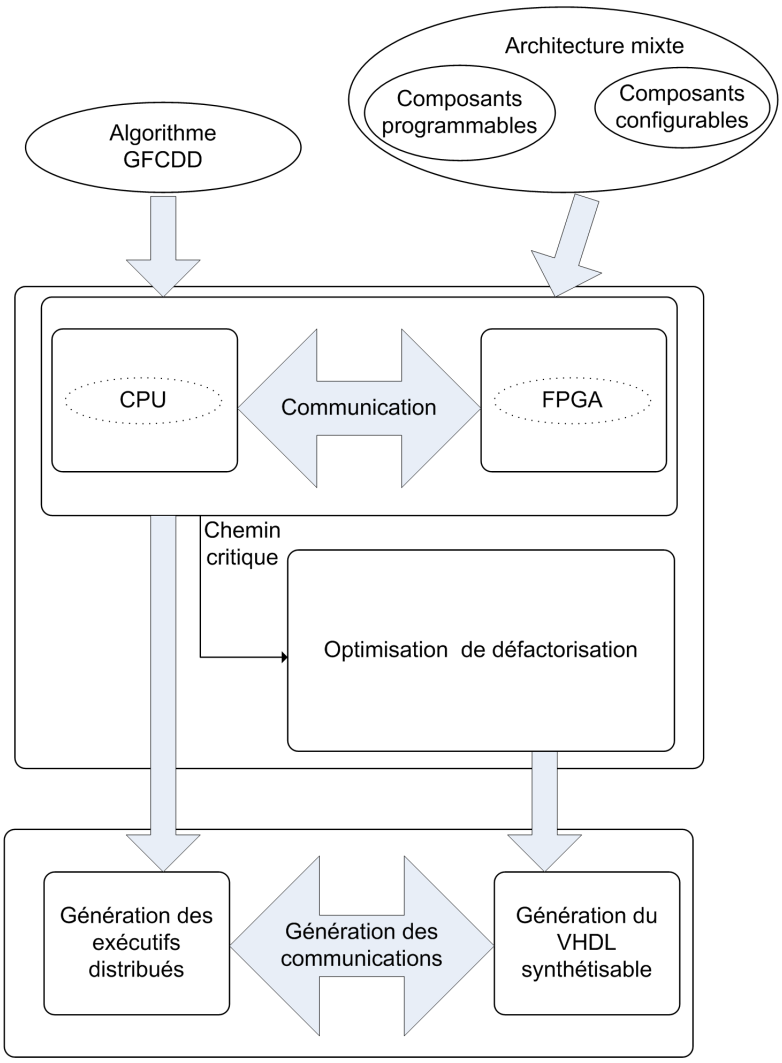

FIGURE 4.1 – Objectif de l'extension de la méthodologie AAA

A l'issue de l'étude du modèle d'architecture utilisé dans la méthodologie AAA (page 38) pour les composants programmables et celui utilisé dans son extension pour les composants reconfigurables (page 50), nous pouvons constater qu'ils ne sont pas compatibles. En effet la méthodologie AAA pour les composants programmables utilise un ensemble de sommets qui ne dépendent pas de l'algorithme et de son implémentation pour modéliser l'architecture. Alors que le modèle d'architecture utilisé par l'extension de AAA pour les architectures reconfigurables est obtenue par traduction directe du graphe d'algorithme qui correspond à l'implémentation optimisée. Le modèle d'architecture de l'extension de AAA est donc déduit à partir de l'implémentation optimisée, alors que celui de AAA pour les architectures programmables est un point de départ pour pouvoir optimiser l'implémentation. Dans ce chapitre, nous proposons une extension du modèle d'architecture permettant de modéliser les composants reconfigurables en utilisant des sommets compatibles avec le modèle d'architecture de AAA. Par la suite, nous présentons l'algorithme de couplage entre les heuristiques de SynDEx et SynDEx-IC pour le partitionnement matériel/logiciel automatique et l'optimisation de l'implémentation des algorithmes sur des architectures mixtes.

4.2 Présentation des composants reconfigurables

On a vu dans le paragraphe 2.2.2 qu'un FPGA est un circuit électronique préfabriqué qui peut être configuré électriquement pour pouvoir exécuter n'importe quelle fonction numérique voulue. Il comporte une quantité importante de ressources qui sont essentiellement : des éléments logiques reconfigurables, un réseaux d'interconnexions reconfigurables et des blocs d'entrée/sortie reconfigurables. Un FPGA peut aussi contenir des mémoires et des opérateurs câblés. Nous allons nous intéresser essentiellement aux éléments reconfigurables des FPGAs.

4.2.1 Ressources logiques reconfigurables

Les blocs logiques reconfigurables (ou **C**onfigurables **L**ogic **B**loc en Anglais) constituent les éléments de base du FPGA. Ils peuvent utiliser différentes technologies. On cite essentiellement les technologies basées sur des portes NAND, des interconnexions de multiplexeurs [Gamal et al., 1989] et des "Look Up Table" (LUT) [Macii and Poncino, 1994]. La technologies des LUT est largement utilisée pour les FPGAs commerciaux fabriquées par les leaders du domaine : Xilinx [Xilinx, a] et Altera [Altera, a]. Ces blocs logiques reconfigurables peuvent être constitués d'un ou plusieurs éléments logiques de base (ELB). Un élément logique de base est constitué, dans la plus part des cas, d'une LUT associée à une bascule. La figure 4.2 représente un élément logique de base constitué d'une LUT à 4 entrées (LUT-4). Cette LUT-4 utilise 16 SRAM d'un bit pour la configuration et peut implémenter n'importe quelle fonction booléenne à 4 entrées. La sortie de cette LUT est connectée à une bascule. Le multiplexeur permet de choisir entre connecter la sortie du ELB à la sortie de la LUT ou à celle de la bascule. Un CLB peut contenir un groupe de ELBs connectés par un réseau local d'interconnexions. La figure 4.3 montre un exemple de CLB composé par quatre ELB. Le nombre de sorties du CLB est égal au nombre de ELB qu'il contient, alors que le nombre de pins d'entrées doit être inférieur ou égal au nombre de ELB que contient le CLB. Les FPGAs modernes comportent de 4 à 10 ELB par CLB.

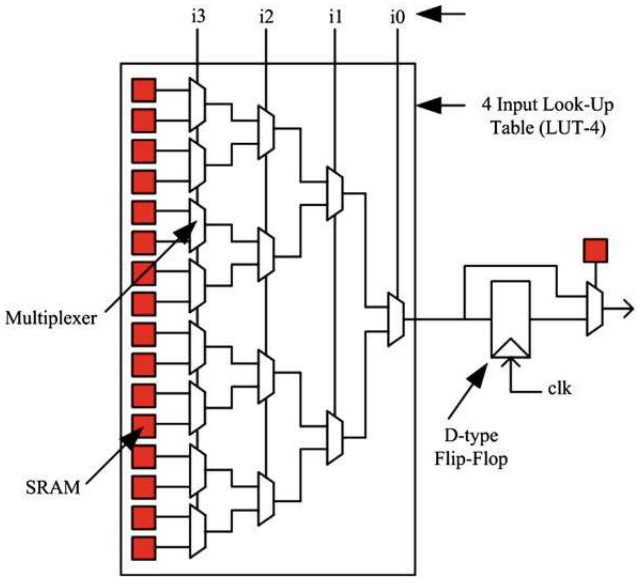

FIGURE 4.2 – Elément logique de base

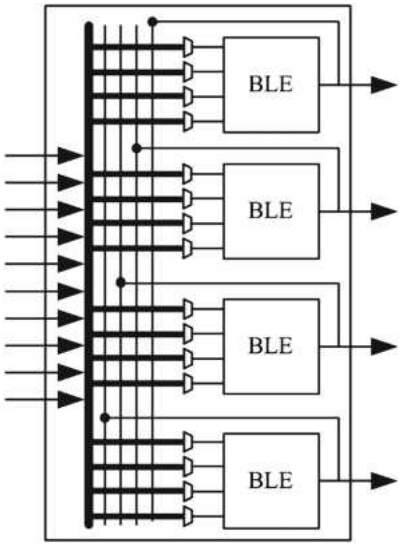

FIGURE 4.3 – Bloc logique reconfigurable constitué par 4 ELB

Le tableau 4.1 présente le nombre d'éléments logiques en (k éléments) et les fréquences maximales de fonctionnement de quelques familles de FPGA proposées par les leadeurs du marché de FPGA.

Tableau 4.1 – Caractéristiques de quelques familles de FPGA

Fabriquant	Famille de FPGA	Nombre d'éléments logiques (en k)	Fréquence de fonctionnement
Altera	Stratix	236-952	420 MHz
	Stratix III	48-338	800 MHz
	Stratix IV	72,6-813	800 MHz
	Stratix V	236-952	800 MHz
	Arria 10	156-1678	800 MHz
	Cyclone V	25-300	550 MHz
Xilinx	Spartan-6	4-147	1 GHz
	Artix-7	33-215	800 MHz
	Kintex-7	65-477	1 GHz
	Virtex-7	326-1954	1 GHz

4.2.2 Architecture du réseau d'interconnexions

L'architecture d'un FPGA peut être décrite par la topologie de son réseau d'interconnexions. Il existe principalement deux types d'architectures de réseaux d'interconnexions utilisées pour les FPGA : l'architecture en îlots de calcul et l'architecture hiérarchique. Il existe une variante combinant ses 2 approches : on utilise une architecture en îlots de calcul et on intègre une hiérarchie au niveau de ces îlots.

Architecture en îlots de calcul

La figure 4.4 représente une schématisation simplifiée de l'architecture îlot de calcul des FPGA. Cette architecture est la plus utilisée pour la conception de FPGA [Farooq et al., 2012]. Elle est appelée ainsi car les CLB sont entourés par le réseau d'interconnexions comme la mer entoure les iles. En effet, dans cette architecture, les CLB sont disposés en matrices. Le réseau d'interconnexions est formé de lignes de routage disposées horizontalement et verticalement autour de ces CLB. Des commutateurs reconfigurables connectent ces lignes de routage entre elles et avec les différents CLB. Les lignes de routage sont de plusieurs longueurs pour réduire les délais de routage.

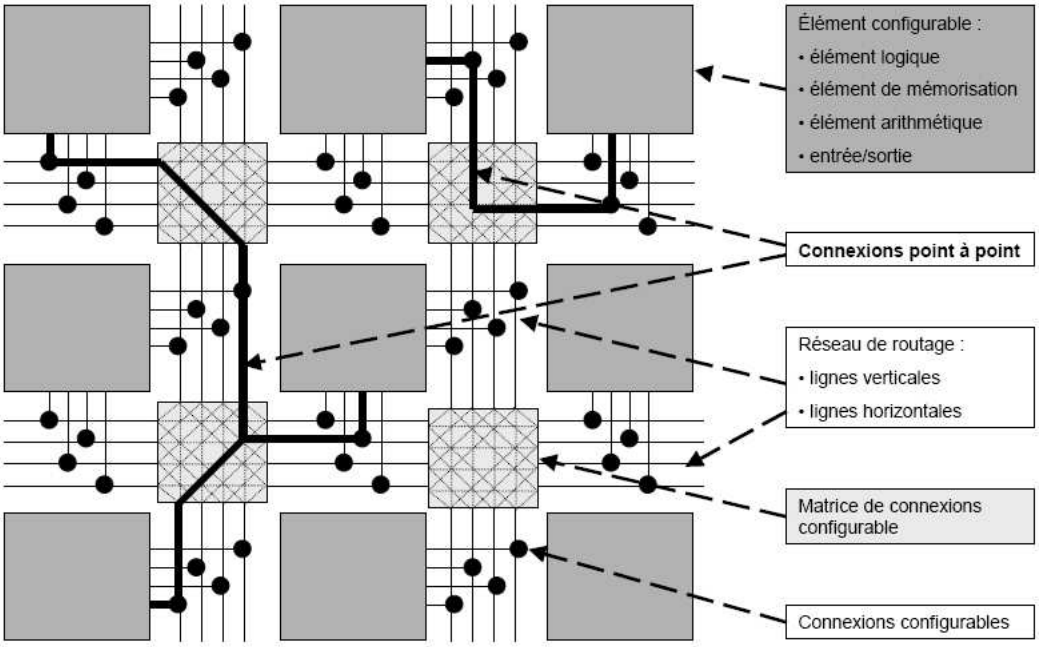

FIGURE 4.4 – Architecture îlots de calcul [Bossuet, 2004]

La figure 4.5 représente un exemple de différentes longueurs de lignes de routage. Les lignes de longueur importante permettent de couvrir plusieurs CLB mais réduisent la flexibilité de routage et par la suite la probabilité de réussir à router plusieurs circuits. Un compromis entre la flexibilité et l'efficacité de routage existe aujourd'hui dans les FPGA modernes en utilisant des lignes de routage de petites, moyennes et grandes longueurs [Farooq et al., 2012].

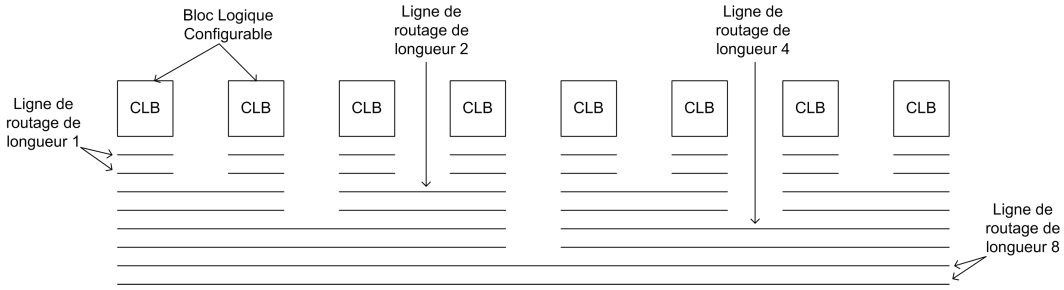

FIGURE 4.5 – Distribution des longueurs des lignes de routage

Architecture hiérarchique

L'architecture hiérarchique des FPGA favorise les communications locales par le regroupement des CLB en “clusters". Chaque cluster contient un réseau local d'interconnexions. Un ensemble de clusters connectés entre eux par un réseau d'interconnexions reconfigurable forme le niveau hiérarchique suivant. Ainsi chaque niveau hiérarchique est formé par les éléments du niveau inférieur connecté par un réseau d'interconnexions reconfigurable. L'architecture hiérarchique couramment utilisée dans les FPGA commerciaux est constituée de trois à quatre niveaux hiérarchiques.

La figure 4.6 représente les différents niveaux hiérarchiques utilisés dans les FPGA ACTEL.

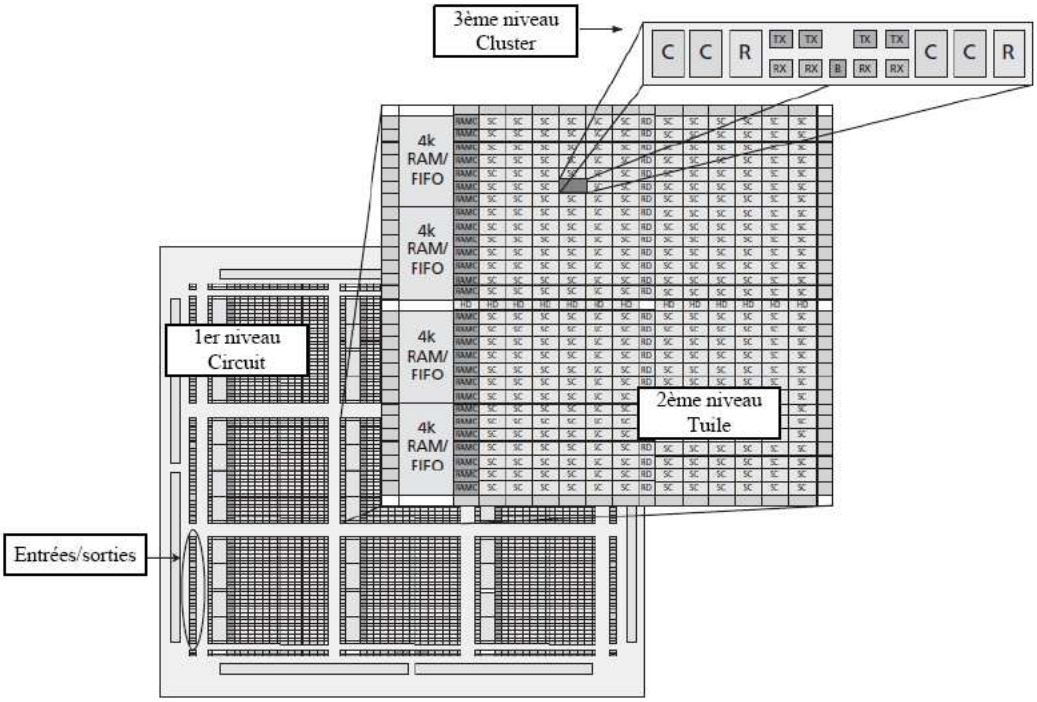

FIGURE 4.6 – Architecture hiérarchique des circuits ACTEL [Actel,]

4.2.3 Modélisation des composants reconfigurables

Comme indiqué en introduction, nous avons besoin d'étendre le modèle d'architecture AAA qui jusqu'à présent ne permet pas de modéliser le parallélisme interne et massif qu'offrent les FPGA. Si l'on souhaite utiliser efficacement ces composants, il faut pouvoir décrire plus finement les ressources internes qu'ils renferment sans descendre dans une modélisation trop fine. Cette modélisations doit être compatible avec le modèle d'architecture de AAA qui repose actuellement sur les sommets opérateurs de calcul, opérateurs de communication, mémoires et bus comme détaillé dans la section 3.3.2.

Nous venons de voir qu'un FPGA est constitué essentiellement de blocs logiques reconfigurables, de blocs de mémoire et de réseaux d'interconnexions reconfigurables. Nous proposons de modéliser chaque bloc logique reconfigurable par un “opérateur élémentaire $OPRe$”.

Nous définissons S_{OPRe} l'ensemble des opérateurs élémentaires non configurés, S_{reg} l'ensemble des registres représentant les blocs mémoire et R le réseau d'interconnexions

reconfigurable.

Ainsi nous pouvons modéliser un FPGA non configuré par un graphe $(S_{OPRe} \cup S_{reg}, R)$

où chaque sommet représente un bloc logique ou de mémorisation et chaque arc représente une ligne d'interconnexion.

4.2.4 Opération de configuration

Nous venons de voir qu'un FPGA ne peut effectuer aucune opération sans configuration. En effet l'opération de configuration s'applique à un sous-ensemble du circuit FPGA en utilisant des outils de synthèse comme par exemple ISE de Xilinx, Quartus II d'Altéra et Libero d'Actel pour les principaux. Cette opération de configuration permet de configurer les blocs logiques pour effectuer des opérations logiques élémentaires (AND, OR, ...). En parallèle, la configuration du réseau d'interconnexions permet d'établir des connexions entre ces blocs configurés pour pouvoir effectuer des opérations complexes. Il existe essentiellement trois méthodes pour configurer un FPGA : la configuration par mémoire statique [Hsieh et al., 1988], la configuration par mémoire flash [Guterman et al., 1979] et la configuration par anti-fusible [Chiang et al., 1992].

La plus part des fabricants des FPGA utilisent la technologie de mémoire statique pour la configuration. Les points forts de cette méthode de configuration sont sa grande vitesse et sa faible consommation dynamique. Mais cette technologie est coûteuse en surface. De plus, les mémoires utilisées pour la configuration sont volatiles, d'où la nécessité de mémoires externes pour sauvegarder en permanence les données de configuration [Farooq et al., 2012].

Une autre alternative pour la configuration des FPGA consiste à utiliser des mémoires flash. Ces mémoires sont non volatiles donc ne nécessitent pas d'autres éléments de stockage. Cette technologie est en plus moins gourmande en surface. Mais les FPGA utilisant cette technologie ne sont pas reconfigurables une infinité de fois [ARM, 2011].

La technologie de configuration des FPGA par anti-fusible ajoute les plus faibles résistances et capacités parasites comparés aux technologies de configuration de FPGA citées précédemment. Mais les FPGA utilisant cette technologie ne sont configurables qu'une seule fois et ne peuvent pas être reconfigurés.

4.2.5 Modélisation de la configuration

Bien que nous n'allons pas nous substituer aux outils de synthèse qui configurent les FPGA, nous avons besoin de modéliser formellement les transformations qu'ils effectuent sur le modèle proposé précédemment. En effet dans cette méthodologie, il est fondamental de pouvoir décrire dans un flot sans rupture toutes les transformations qui mènent de la spécification de l'algorithme et de l'architecture jusqu'à l'exécution de l'application. Nous modélisons donc l'opération de configuration d'un FPGA par la relation ρ. Cette configuration est appliquée aux opérateurs élémentaires. Il en résulte un opérateur élémentaire configuré capable d'exécuter une opération logique que l'on qualifiera d'opération logique élémentaire.

Une fois configuré, la fonction associée à un opérateur élémentaire ne peut être modifiée que par une reconfiguration. Un ensemble de ces opérateurs élémentaires configurés,

ainsi qu'une partie des registres configurés, peuvent être connectés entre eux par le réseau d'interconnexions configuré de manière à ce que l'ensemble puisse effectuer des opérations plus complexes. L'opération de configuration peut aussi configurer un ensemble de registres contenus dans le FPGA et les connecter à travers le réseau d'interconnexions configuré pour produire des mémoires. Ces mémoires peuvent être allouées pour sauvegarder les données à traiter et les résultats. On rappelle que ces mémoires sont contenues dans des composants reconfigurables qui ne nécessitent pas une programmation, donc elles ne peuvent pas être utilisées pour sauvegarder des instructions.

La fonction de configuration ρ s'applique à un FPGA pour produire des mémoires et des opérateurs capable chacun d'exécuter un seul type d'opération du graphe d'algorithme. Ces opérateurs seront nommés **opérateurs dégénérés** dans la suite de ce document.

Après configuration, le modèle du FPGA comporte un ensemble d'opérateurs dégénérés (S_{OPRd}) et un ensemble de mémoires S_M ainsi que les opérateurs élémentaires non configurés, des registres non configurés sans oublier la partie du réseau d'interconnexions non configurée qui resteront donc inutilisés.

La fonction de configuration du FPGA ρ est définie par :

$$\rho : S_{OPRe} \cup S_{reg} \cup R \longrightarrow S_{OPRd} \cup S_M \cup S'_{OPRe} \cup S'_{reg} \cup R'$$

$$\text{Avec } S'_{OPRe} \subset S_{OPRe}, S'_{reg} \subset S_{reg} et R' \subset R.$$

S'_{OPRe}, S'_{reg} et R' représentent respectivement l'ensemble des opérateurs élémentaires, l'ensemble des registres et la partie du réseau d'interconnexions non configurés après l'obtention de l'ensemble des opérateurs dégénérés S_{OPRd}.

La figure 4.7 schématise un FPGA avant configuration. Dans ce schéma le FPGA est constitué de blocs logiques reconfigurables, des registres et le réseau d'interconnexions. La figure 4.8 représente l'algorithme à implémenter dans le FPGA. Cet algorithme est constitué de deux opérations ayant chacune deux entrées et une sortie. Le FPGA configuré est représenté par la figure 4.9, il contient deux opérateurs dégénérés : OPRd 1 formé à partir des opérateurs élémentaires 1 et 2 et OPRd formé à partir des opérateurs élémentaires 7 et 8 comme présente le tableau 4.2. Dans cette représentation (figure 4.9) les lignes du réseau d'interconnexions utilisées ainsi que les connexions établies sont représentées en traits rouges épais.

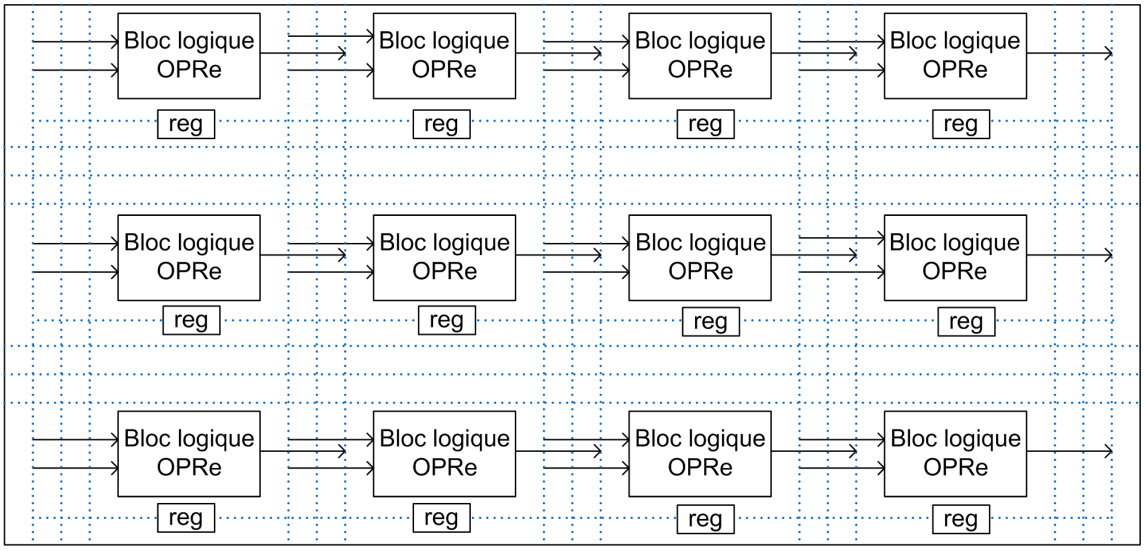

FIGURE 4.7 – Représentation simplifiée de la configuration du FPGA

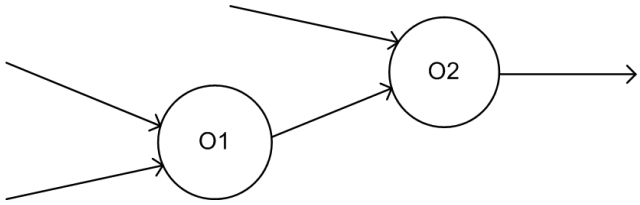

FIGURE 4.8 – Représentation de l'algorithme à implémenter

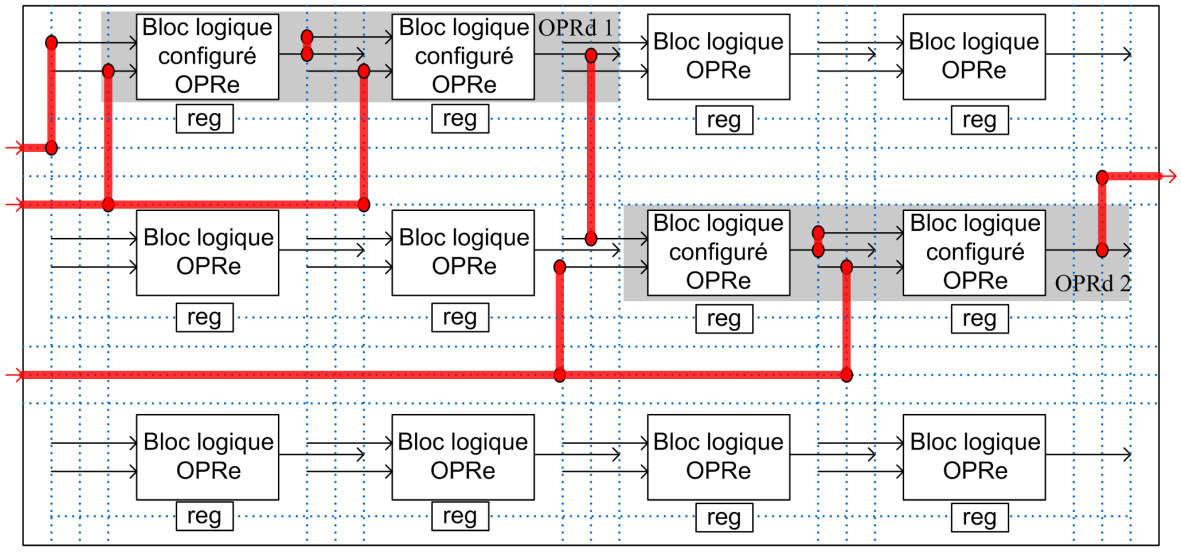

FIGURE 4.9 – Représentation simplifiée du FPGA après configuration

Tableau 4.2 – Composants du FPGA utilisés

Composant avant configuration	Composant après configuration
$OPRe_1$ $OPRe_2$	$OPRd_1$
$OPRe_7$ $OPRe_8$	$OPRd_2$

La partie gauche de la figure 4.10 présente l'ensemble des sommets que contient le FPGA avant configuration. Il est composé par un ensemble d'opérateurs élémentaires, de registres et un réseau d'interconnexion. La partie droite de la figure 4.10 représente le FPGA après configuration. On y constate bien l'apparition d'un ensemble d'opérateurs dégénérés utilisables pour implanter les opérations de l'algorithme représenté par le rectangle et les ressources inutilisées, donc non reconfigurées, représentées par un ensemble en pointillé.

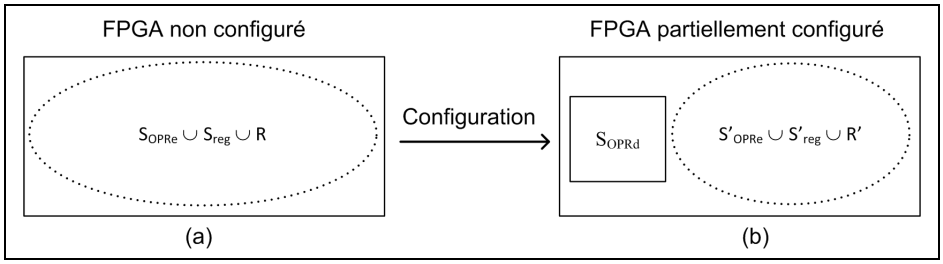

FIGURE 4.10 – Représentation du FPGA : (a) avant configuration, (b) après configuration

4.3 Modélisation des communications intra composant reconfigurable

Les différents opérateurs dégénérés contenus dans un FPGA sont connectés entre eux à travers le réseau d'interconnexions configurable qui permet d'établir des liaisons multiples et permet aussi la diffusion matérielle des données à plusieurs opérateurs. Ce réseau peut être modélisé par une SAM (voir page 39) multi-point permettant la diffusion matérielle des données. Au niveau qui nous intéresse, le temps de traversé des fils électriques qui composent ce réseau d'interconnexions est suffisamment faible et le niveau d'abstraction de notre modèle est suffisamment haut pour négliger le temps de communication à l'intérieur d'un composant par rapport au temps de traitement. La durée de traversé de cette SAM quelque soit le type de données sera alors considérée comme nulle. Pour cette raison, ces SAM ne sont pas représentées dans les graphes d'implémentation. Par conséquence, puisque ces sommets ne sont pas représentées, les connexions entre les différents opérateurs d'un FPGA dans le graphe d'implantation seront modélisées par de simples arcs.

4.4 Modélisation des communications avec les composants reconfigurables

Jusqu'ici nous n'avons traité que des communications entre les opérateurs dégénérés appartenant à un même FPGA : ces communications utilisent le réseau d'interconnexions du FPGA. Cependant, nous devons aussi pouvoir modéliser les communications entre des opérateurs dégénérés et d'autres opérateurs contenus dans les autres composants de l'architecture. Comme pour les communications entre processeurs, il existe deux grands types de communications :

1. connexion par un bus de communication de type FIFO (SAM)
2. connexion par mémoire partagée (RAM)

Puisque cette communication doit suivre le modèle de communication AAA, nous devons aussi utiliser un communicateur (voir page 40) coté FPGA. Il sera appelé par la suite "IP de communication".

Celui ci sera réalisé par la configuration d'un ensemble de ressources matérielles. Il permettra donc de séquencer des opérations de communication entre les processeurs et le FPGA ou entre les différents FPGA.

4.5 Modélisation du FPGA

L'objectif de la modélisation du FPGA est de mettre en évidence les ressources allouables qu'il renferme afin de pouvoir lui faire exécuter le graphe d'algorithme. Ces ressources sont les communicateurs et les opérateurs dégénérés obtenus par la configuration. Vis à vis du modèle d'architecture AAA existant, on peut considérer qu'un FPGA est constitué d'un ensemble d'opérateurs et communicateurs tous connectés entre eux (graphe connexe) parmi lequel seront choisis et alloués ceux nécessaires pour l'implémentation du graphe d'algorithme. Cela sera décrit dans le prochain paragraphe.

La figure 4.11 représente le graphe d'architecture d'un FPGA avant allocation.

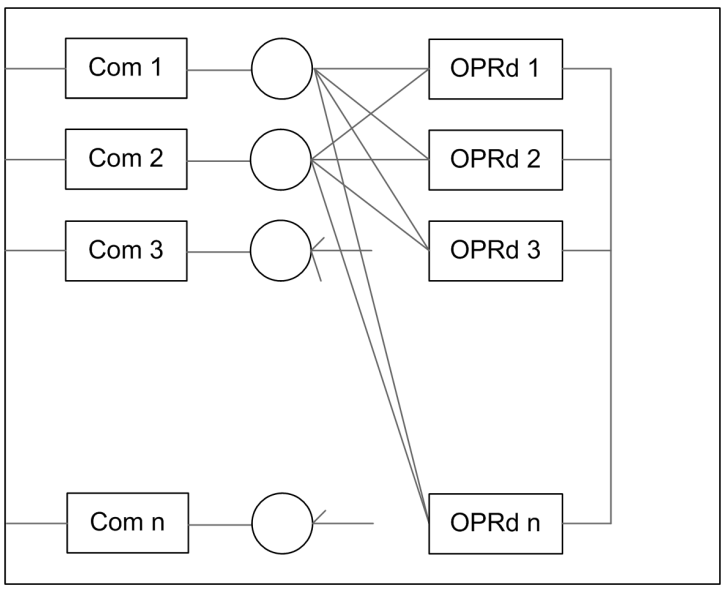

FIGURE 4.11 – Représentation fonctionelle du FPGA avant configuration

Etant donné qu'après allocation des opérateurs, un certain nombre d'opérateurs dégénérés apparaitront comme non utilisés, il est possible de simplifier la représentation de ce graphe d'architecture FPGA en ne pas montrant les opérateurs dégénérés inutilisés. Cette simplification conduit à un sous-graphe composé par un opérateur dégénéré pour chaque opération qu'il est capable d'exécuter et un IP de communication pour chaque port de connexion entre le FPGA et les autres composants de l'architecture. Ces opérateurs dégénérés et IPs de communication sont connectés entre eux à travers une SAM multi-points permettant la diffusion matérielle des données. Cette SAM modélise le réseau d'interconnexions, les opérateurs dégénérés modélisent les opérations que peut exécuter le FPGA et les IPs de communication modélisent l'aspect séquentiel de communication à travers chaque port du FPGA. Comme précisé préalablement, la durée de communication à travers la SAM modélisant le réseau d'interconnexions pour tout type de données est nulle car on se place à un niveau d'abstraction beaucoup plus élevé. Par contre, les communications à travers les IPs de communication ont des durées non nulles car elles nécessitent une synchronisation.

Un FPGA configuré et alloué pour exécuter trois opérations et ayant un seul port de communication, peut être modélisé par la figure 4.12. On y distingue un seul IP de communication (com1) et trois opérateurs dégénérés (OPRd1, OPRd2, OPRd3).

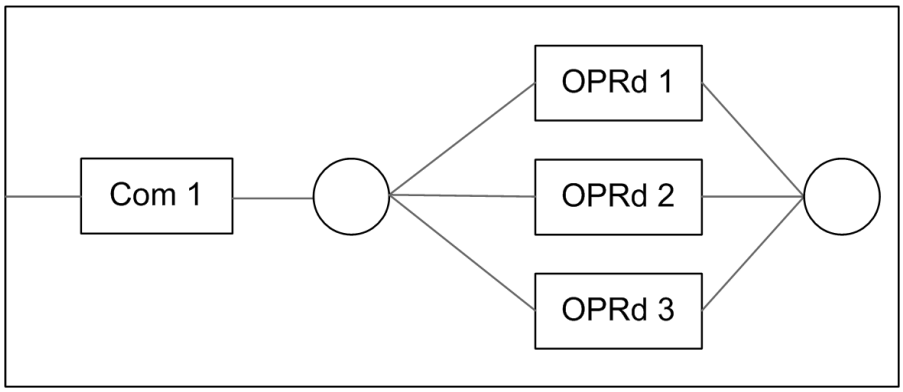

FIGURE 4.12 – Exemple de représentation fonctionelle du FPGA configuré

4.6 Extention du modèle d'implémentation

Comme nous l'avons vu précédemment, un FPGA doit être configuré pour réaliser une ou plusieurs fonctions numériques (section 4.2). Le FPGA configuré est alors capable d'exécuter une ou plusieurs opérations du graphe d'algorithme. Si le FPGA exécute plusieurs opérations du graphe d'algorithme qui ne sont pas en dépendance de données, ces opérations pourront être exécutées simultanément. En effet, ces opérations pourront être implantées par des opérateurs dégénérés indépendants. Ainsi le calcul des dates d'exécution des opérations implémentées par des opérateurs dégénérés est modifié par rapport au calcul des dates des opérations implantées par des opérateurs des architectures programmables.

Rappelons que dans le cas d'implémentation sur un composant programmable, la date de début au plus tôt se calcule en utilisant l'équation 4.1 et la date de fin au plus tard se calcule en utilisant l'équation 4.2. Notons que ces deux équations tiennent compte respectivement des prédécesseurs et des successeurs de l'opération distribués sur le même

opérateur. En effet, l'exécution d'une opération sur un opérateur programmable ne peut débuter que si cet opérateur est libre (n'est pas en train d'exécuter une autre opération).

$$S(O_i) = \begin{cases} 0 & \text{si } \Gamma^{-1}(O_i) = \Gamma'^{-1}(O_i) = \emptyset \\ \max(\max_{\forall O_j \in \Gamma^{-1}(O_i)} E(O_j), E(\Gamma'^{-1}(O_i))) & \text{sinon} \end{cases} \quad (4.1)$$

Avec $E(O_i)$ est la date de fin au plus tôt de l'opération O_i, $\Gamma'^{-1}(O_i)$ l'ensemble de ses prédécesseurs distribués sur le même opérateur et $\Gamma^{-1}(O_i)$ l'ensemble des prédécesseurs de O_i.

$$\bar{E}(O_i) = \begin{cases} 0 & \text{si } \Gamma(O_i) = \emptyset \text{ et } \Gamma'(O_i) = \emptyset \\ \max(\max_{\forall O_j \in \Gamma(O_i)} \bar{S}(O_j), \bar{S}(\Gamma'(O_i))) & \text{sinon} \end{cases} \quad (4.2)$$

Avec $\bar{S}(O_i)$ est la date de début au plus tard de l'opération O_i, $\Gamma'(O_i)$ l'ensemble de ses successeurs distribués sur le même opérateur et $\Gamma(O_i)$ l'ensemble de ses successeurs de O_i.

Dans le cas d'implémentation d'une opération sur un opérateur reconfigurable, l'équation de calcul de la date de début au plus tôt devient l'équation 4.3 et celle de calcul de la date de fin au plus tard devient l'équation 4.4. Ces nouvelles équations ne tiennent pas compte de la disponibilité de l'opérateur reconfigurable qui exécute l'opération. Puisque ces composants reconfigurables peuvent être configurés pour contenir plusieurs opérateurs fonctionnant en parallèle.

$$S(O_i) = \begin{cases} 0 & \text{si } \Gamma^{-1}(O_i) = \emptyset \\ \max_{\forall O_j \in \Gamma^{-1}(O_i)} E(O_j))) & \text{sinon} \end{cases} \quad (4.3)$$

$$\bar{E}(O_i) = \begin{cases} 0 & \text{si } \Gamma(O_i) = \emptyset \\ \max_{\forall O_j \in \Gamma(O_i)} \bar{S}(O_j) & \text{sinon} \end{cases} \quad (4.4)$$

Dans l'exemple suivant, nous allons montrer l'impact de notre extension du modèle AAA sur le calcul de dates qui sera ainsi conforme à la réalité. Pour cela, nous présentons l'implantation d'un algorithme (figure 4.13) sur une architecture constituée d'un processeur mono-cœur et d'un FPGA modélisé par le modèle AAA non étendu puis l'implémentation du même algorithme sur la même architecture modélisée par notre extension. Dans les deux cas, on impose que le sous-graphe d'algorithme entouré de pointillé (opérations O2 et O3) soit implanté sur le FPGA du graphe d'architecture et le reste (opérations O1 et O4) sur le processeur.

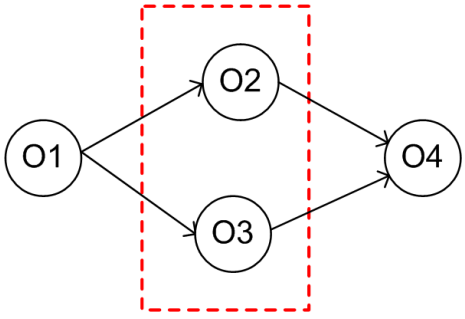

FIGURE 4.13 – Exemple de graphe d'algorithme

Tout d'abord, la figue 4.14 modélise l'architecture en utilisant le modèle AAA sans extension. Ce graphe d'architecture contient un processeur comportant un opérateur et un communicateur connectés à travers une RAM. Un sous-graphe identique est utilisé pour modéliser le FPGA. Le processeur et le FPGA communiquent à travers une mémoire SAM.

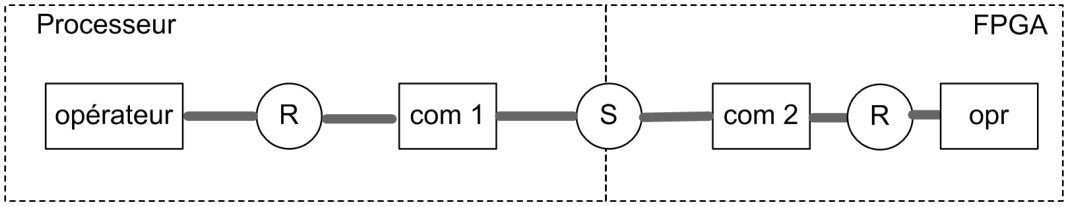

FIGURE 4.14 – Graphe d'architecture sans extension de AAA

Comme présenté à la page 43, l'opération d'implémentation se décompose en trois parties : routage, partitionnement et ordonnancement. Le routage établit les chemins de communication entre les différents opérateurs du graphe d'architecture. Le partitionnement affecte à chaque opérateur les opérations qu'il exécute. L'ordonnancement établit l'ordre d'exécution entre les opérations. l'implémentation ne modifie pas les graphes d'algorithme et d'architecture. La figure 4.15 donne un exemple de graphe d'implantation obtenue en utilisant l'heuristique SynDEx. Comme on s'y attendait, cette implémentation ne peut pas tenir compte du parallélisme du FPGA car il place O2 et O3 en séquentiel alors qu'ils peuvent être placés en parallèle sur FPGA. Sur cette figure, la hauteur des rectangles des opérations de calcul et de communication correspond à leurs durées d'exécution. Par conséquent D1 est la latence de l'algorithme.

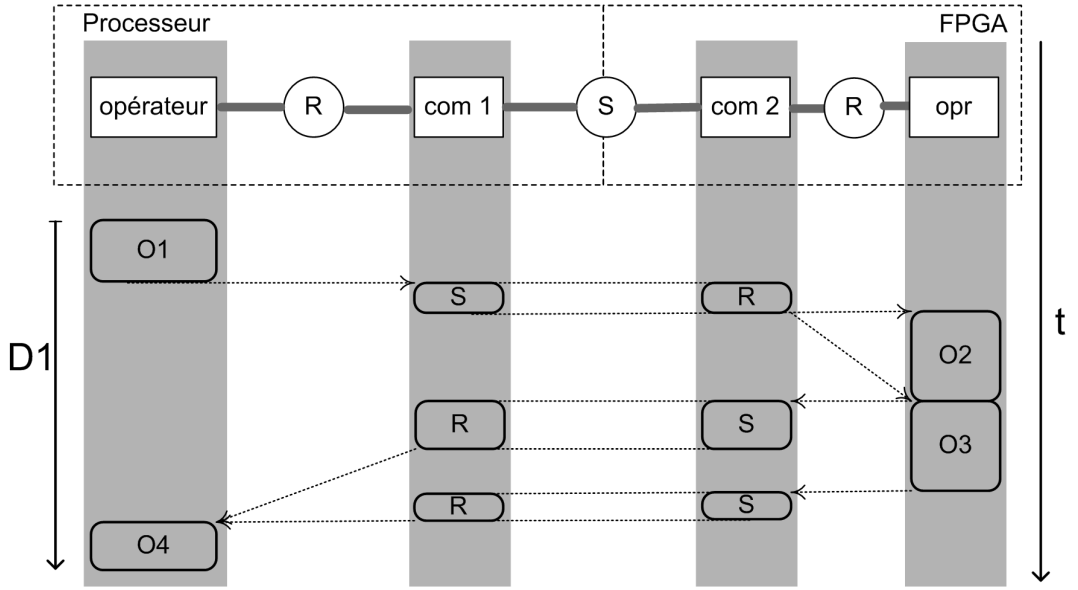

FIGURE 4.15 – Graphe d'implantation en utilisant le modèle d'implantation initial

Voyons maintenant l'impact de notre extension sur ce même exemple. En utilisant notre extension du modèle AAA, l'architecture cible est modélisée par le graphe de la figure 4.16. Le sous-graphe modélisant le processeur n'est pas modifié. Par contre, on tient compte du fait que le FPGA n'est initialement pas configuré.

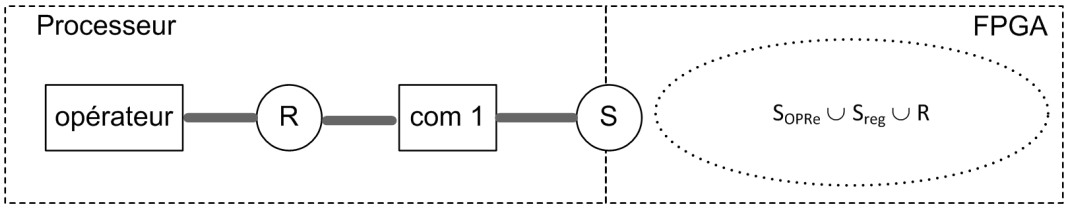

FIGURE 4.16 – Graphe d'architecture en tenant compte de l'extension de AAA pour les architectures mixtes

L'opération de configuration s'effectue au moment de l'implémentation qui se décompose désormais en quatre opérations : routage, configuration, partitionnement et ordonnancement. Cette opération d'implémentation s'applique sur le couple graphe d'algorithme, graphe d'architecture (Gal,Gar) et produit le couple formé par le graphe d'algorithme initial et un nouveau graphe d'architecture contenant les FPGA configurés (Gal,Gar'). Puisque, dans notre exemple le FPGA est capable d'exécuter les opérations O2 et O3, on retrouve dans le modèle du FPGA configuré deux opérateurs dégénérés en plus de l'IP de communication (IP_com). La figure 4.17 représente le graphe d'implantation obtenue. On remarque, qu'en utilisant notre extension du modèle d'architecture, le graphe d'implémentation obtenu tient compte du parallélisme offert par l'utilisation du FPGA. Ainsi les deux opérations O2 et O3 sont implémentés en parallèle sur le FPGA. Ceci permet de réduire la latence de l'algorithme par rapport à l'implémentation précédente : on remarque bien que la latence D2 de cette version est inférieure à la latence D1 de la version précédente.

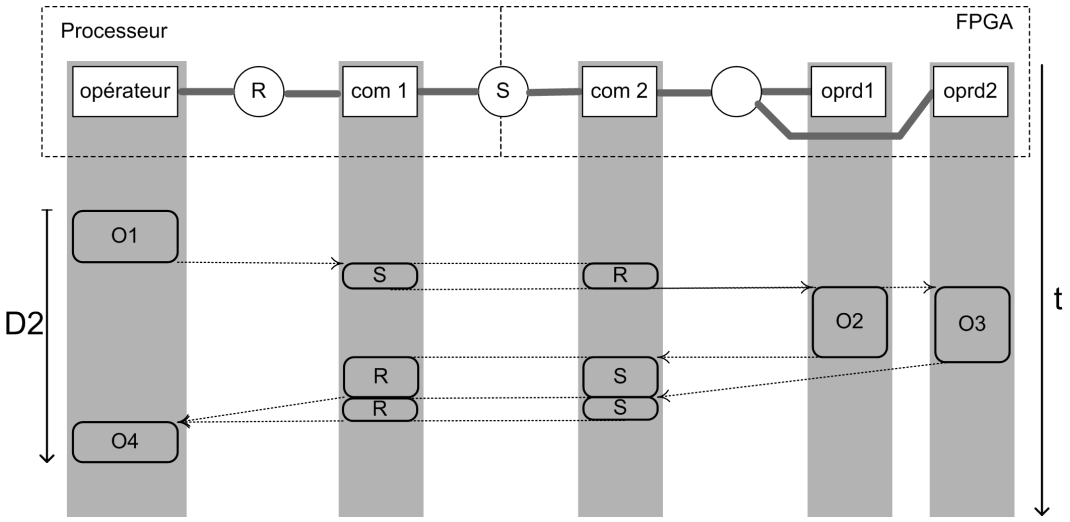

FIGURE 4.17 – Graphe d'implantation obtenu après extension du modèle

4.7 Etat de l'art des algorithmes de partitionnement pour le co-design

Nous avons maintenant étendu le modèle d'architecture et d'implantation de AAA pour couvrir les architectures mixtes. Il nous faut maintenant étudier l'optimisation du partitionnement et de l'allocation de l'algorithme sur l'architecture. Pour cela, nous commençons par l'état de l'art des techniques existantes avant de présenter la notre.

La recherche dans le domaine de co-design vise à développer des méthodologies et des outils permettant le développement simultané des parties matérielles et logicielles tout en tenant compte des interactions entre ces deux parties du système. Ainsi, ces recherches couvrent plusieurs aspects du processus de conception : la spécification et modélisation du système, la co-simulation, le partitionnement et la génération d'interfaces. Nous allons nous intéresser dans cette partie du document aux algorithmes de partitionnement logiciel/matériel.

4.7.1 Le projet ECOS

Le projet ECOS [Freund et al., 1997] tente de fournir un environnement pour le co-design. L'objectif de ECOS est de minimiser le coût des systèmes tout en respectant une contrainte temporelle. L'architecture cible est constituée par un processeur et un ASIC.

L'application est représentée par un graphe direct acyclique où chaque nœud représente une opération de communication ou de traitement de données et chaque arc représente une dépendance de données. Chaque nœud est caractérisé par un coût d'implémentation matériel et un coût d'implémentation logiciel. L'algorithme de partitionnement utilise une adaptation de l'algorithme d'ordonnancement à force dirigée (force-directed scheduling algorithm) au partitionnement matériel/logiciel [Rousseau et al., 1995]. C'est un algorithme itératif qui pour chaque itération, calcule le coût des implémentations possibles pour chaque nœud et choisit celui correspondant au coût minimal et itère l'opération tandis qu'il reste des nœuds pas traités.

Cet algorithme de partitionnement n'est pas adapté aux applications complexes car il prend un temps énorme pour donner le résultat. Il n'est pas non plus adapté pour les applications ayant une forte parallélisation potentielle à cause de la non pertinence de l'estimation de l'impact de l'ordonnancement d'un nœud sur les autres nœuds quand un haut degré de parallélisme est présent.

4.7.2 Méthode de Eles et al.

Eles et al. [Eles et al., 1996] présentent une méthode de partitionnement matériel/logiciel pour un processeur et un coprocesseur matériel. L'objectif de cet algorithme est d'optimiser au maximum la vitesse d'exécution. Pour atteindre cet objectif, l'algorithme de partitionnement tend à augmenter le parallélisme tout en réduisant la communication entre la partie matérielle et la partie logicielle.

Le partitionnement se fait en quatre étapes :

* Spécification du système : l'application est décrite comme un ensemble de processus communicant par passage de messages. Le langage VHDL est utilisé pour cette description [Eles et al., 1994a].
* Extraction des blocs critiques : les blocs de traitement responsables de la majeure partie du temps d'exécution de chaque processus sont extraits. Pour chaque bloc

critique identifié, un nouveau processus est créé ainsi que les canaux de communication avec le processus parent [Eles et al., 1994b].

* Partitionnement de graphe : la spécification de l'application contenant les processus décrits par l'utilisateur ainsi que ceux créés dans l'étape d'extraction des blocs critiques est transformée en un graphe où chaque nœud représente un processus et chaque arc représente un canal de communication. A chaque nœud est associé un poids qui reflète l'aptitude à l'implantation matérielle du processus correspondant. Le poids associé à chaque arc indique une mesure de la communication et la synchronisation entre les processus. Puis un algorithme de recherche tabou est utilisé pour partitionner l'ensemble des processus en deux groupes : un correspondant aux processus qui seront implémentés en logiciel et l'autre à ceux qui seront implémentés en matériel.
* Fusion de processus : les processus créés pendant l'étape d'extraction des blocs critiques qui peuvent être fusionnés avec leurs processus parent s'ils appartiennent à la même partition.

La méthode de Eles et al. ne permet d'effectuer le partitionnement que sur un seul type d'architectures. En effet, elle cible les architectures composées d'un processeur connecté à un coprocesseur matériel. De plus, elle n'effectue pas une exploration automatique de l'espace des implémentations matérielles possibles des parties de l'application implémentée sur le coprocesseur.

4.7.3 Méthode de Zaho et al.

Zaho et al [Zhao et al., 2013] présentent un algorithme de partitionnement matériel/logiciel automatique basé sur une combinaison de recuit simulé et d'algorithme génétique. Etant donnée une contrainte temporelle, cet algorithme tend à trouver un partitionnement qui la respecte en minimisant le coût matériel.

L'application est représentée sous forme de graphe de communication dont chaque nœud représente une tâche principale de l'application et chaque arc représente une dépendance causale ou liée à la transmission de données entre les tâches de l'application. Chaque nœud est caractérisé par son coût d'implémentation matériel, son coût d'implémentation logiciel, sa durée d'exécution matérielle et sa durée d'exécution logicielle. Chaque arc est caractérisé par une durée de communication.

L'algorithme du recuit simulé peut facilement échouer dans une solution qui correspond à un optimum local. Alors que l'algorithme génétique a une capacité de recherche générale forte. Par conséquent, ils ont intégré l'algorithme du recuit simulé dans l'algorithme génétique pour réduire le risque d'échouer dans un minimum local tout en profitant des performances de ces deux algorithmes. Les résultats montrent que l'algorithme combiné offre une solution quasi-optimale plus précise avec une vitesse plus rapide.

La méthode de partitionnement de Zaho et al. vise le partitionnement matériel/logiciel des applications sur une architecture composée d'un unique processeur connecté à un coprocesseur, donc ne permet pas de cibler les architectures multi-processeurs connectées à des coprocesseurs. Elle ne permet pas aussi de chercher une implémentation optimisée des parties distribuées sur le coprocesseur.

4.7.4 Méthode de Han et al.

Han et al. [Han et al., 2013] proposent un algorithme de partitionnement ordonnancement d'algorithmes sur architectures mixtes. Cet algorithme effectue le partitionnement de l'algorithme sur l'architecture multiprocesseur, puis améliore les performances temporelles de l'implémentation en transférant quelques tâches sur la partie matérielle.

L'application est décrite par un graphe de flot de données où chaque nœud représente une tâche et chaque arc représente une dépendance de données entre deux tâches. Chaque nœud est caractérisé par son temps d'exécution logiciel, son temps d'exécution matérielle et la surface nécessaire pour son implémentation matérielle. Les arcs sont caractérisés par le coût de communication qu'il représente.

Cet algorithme est composé de deux parties distinctes. Une heuristique basée sur une liste pour effectuer le partitionnement ordonnancement des tâches du graphe sur les différents processeurs que contient l'architecture cible. La fonction coût utilisée tient compte de la communication inter processeurs. Puis de manière itérative, la tâche du chemin critique qui présente le plus grand rapport gain/surface, est implémentée en matériel jusqu'à ne plus avoir de surface disponible.

Les inconvénients de cette approche est qu'elle ne tient pas compte du coût de communication entre les parties matérielles et logicielles et n'effectue pas une optimisation des parties matérielles.

4.7.5 Méthode de Srinivasan et al.

Srinivasan et al. [Srinivasan et al., 1998] proposent une méthode de partitionnement matériel/logiciel et d'exploration d'espace d'implantation matérielle. L'architecture cible est constituée par un processeur, un coprocesseur et une mémoire partagée par ces deux éléments pour effectuer la communication.

L'application est décrite par un graphe de tâches dont chacune est constituée d'un seul thread et ne peut pas être divisée. Chaque tâche peut représenter une opération (grain fin) ou un bloc fonctionnel (gros grain). Ces tâches sont reliées par des arcs qui représentent les dépendances de données. Pour chaque tâche est spécifiée une table d'implémentations matérielles possibles. La taille de cette table est fixée pour toutes les tâches. Un algorithme génétique est utilisé pour effectuer le partitionnement matériel/logiciel tout en respectant une contrainte temporelle et de surface disponible. Chaque tâche a la même probabilité d'être implémentée en matériel ou en logiciel.

L'inconvénient majeur de cette méthode de partitionnement et qu'elle ne cible qu'un seul type d'architecture. En effet, elle ne permet le partitionnement matériel/logiciel que sur une architecture composée par un processeur et un coprocesseur communicant à travers une mémoire partagée.

4.7.6 Synthèse de l'état de l'art des algorithmes de partitionnement pour le codesign

La plupart des outils et algorithmes de partitionnement logiciel/matériel rencontrés dans la littérature cible une architecture composée par un processeur mono-cœur et un coprocesseur matériel. Cette restriction de l'architecture ne suit pas la tendance actuelle d'exploiter le parallélisme offert par les architectures multi-cœur et multi-processeur en leur ajoutant un ou plusieurs coprocesseurs matériels. De plus, peu de ces outils permettent d'explorer l'espace des solutions pour les parties implémentées en matériel. En

effet, comme expliqué dans la figure 3.14, l'implémentation matérielle peut être complètement séquentielle, complètement parallèle ou une des implémentations intermédiaires entre ces deux cas extrêmes.

Pour combler ce vide, nous proposons une méthode de partitionnement logiciel/matériel et d'optimisation d'implémentation sur les architectures mixtes basée sur la méthodologie AAA. Cette proposition ne fait pas de restrictions sur le nombre de composants programmables ou reconfigurables utilisés. Elle permet, en plus du partitionnement, d'optimiser l'implémentation des parties matérielles pour réduire la latence totale de l'exécution de l'algorithme tout en utilisant le moins possible de blocs logiques.

4.8 Algorithme d'optimisation proposé

L'algorithme que nous proposons repose sur la méthodologie AAA de base et sur son extension : AAA pour les architectures programmables (heuristique de SynDEx) et AAA pour les composants reconfigurables (heuristique de SynDEx-IC).

Le principe global repose sur les actions suivantes. Il est résumé sur la figure 4.18 et détaillé sous la forme d'un algorithme page 80 (algorithme 3).

L'ensemble de ces transformations seront ensuite appliquées sur un exemple concret et complet de la norme H.264.

1. Tout d'abord l'algorithme et l'architecture sont modélisés classiquement dans SynDEx en utilisant le formalisme AAA. Dans le graphe d'architecture, les opérateurs correspondant à des FPGA devront être identifiables par la suite (les détails sont donnés page 81). Nous spécifions ainsi un graphe d'architecture composé d'opérateurs de type FPGA ou non.

   Il faut aussi classiquement caractériser chaque opérateur en donnant la liste des opérations qu'il est capable d'exécuter et leurs durées d'exécution sur cet opérateur (ainsi que le nombre de CLB si c'est un opérateur FPGA).
2. Ensuite nous transformons automatiquement le graphe d'architecture pour faire apparaitre explicitement le parallélisme interne de chaque FPGA : chaque opérateur connu pour être un FPGA est remplacé par un sous-graphe FPGA tel que présenté en 4.5. Ce sous-graphe est fait d'un communicateur (IP de communication) connecté à autant d'opérateurs dégénérés qu'il y a d'opérations exécutables sur cet FPGA. C'est le rôle de la fonction "Transformation"(ligne 3 de l'algorithme 3). Il sera donné en détails page 84 (algorithme 4).
3. L'heuristique AAA, modifiée pour les calculs des dates en 4.6 est appliquée sur le couple graphe d'algorithme, graphe d'architecture transformé. De cette façon SynDEx effectue le partitionnement en tenant en compte toutes les caractéristiques de l'architecture.
4. Après avoir effectué le partitionnement, SynDEx-IC est utilisé pour optimiser l'implantation des parties distribuées sur les FPGA de l'architecture. Le principe de cette optimisation est de réduire la latence des opérations tant qu'elles appartiennent au chemin critique. Cette optimisation n'est effectuée que si le résultat du partitionnement contient au moins une opération distribuée sur FPGA et appartenant au chemin critique. L'optimisation effectuée par SynDEx-IC se déroule en deux étapes séparées par un appel à SynDEx pour recalculer les nouvelles dates de début et de fin des opérations de l'algorithme en utilisant les nouvelles durées calculées par SynDEx-IC.

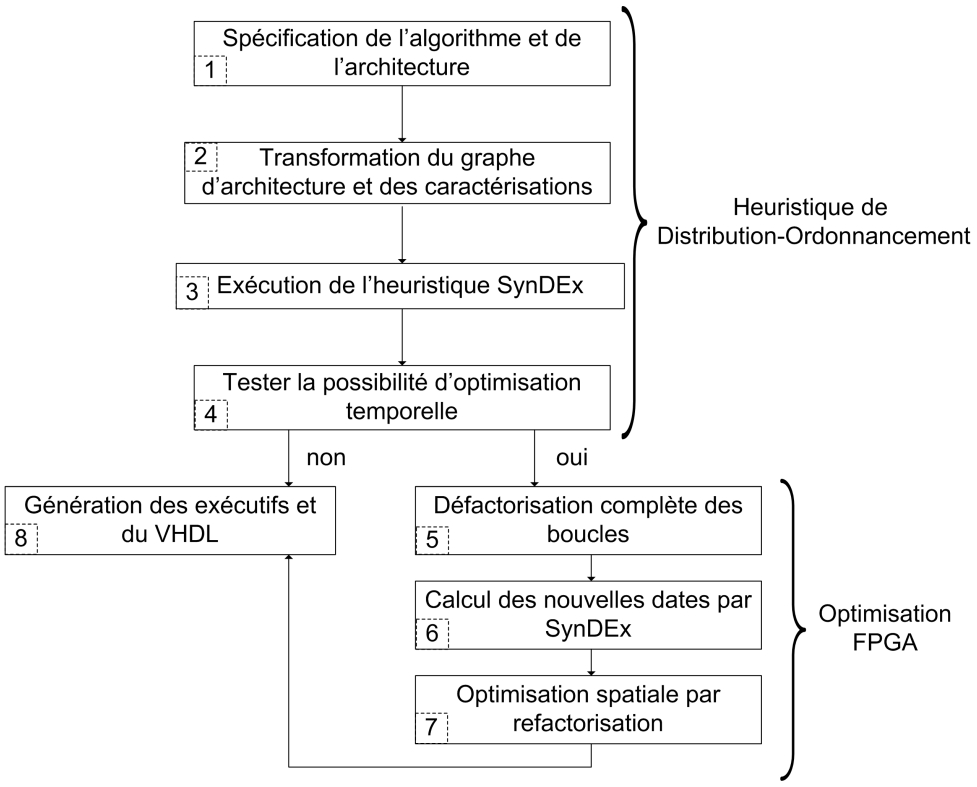

FIGURE 4.18 – algorithme de couplage des outils SynDEx/SynDEx-IC

Algorithm 3 Algorithme de couplage des heuristiques SynDEx/SynDEx-IC)

1: $G_{AR} = \{Opr_1, \ldots, Opr_n, FPGA_1, \ldots, FPGA_m, med_1, \ldots, med_p\}$
2: $G_{AL} = \{O_1, O_2, \ldots, O_q, Arc_1, Arc_2, \ldots, Arc_k\}$
3: $G'_{AR} = Transformation(G_{AR})$
4: Heuristique SynDEx (G_{AL}, G'_{AR})
5: **if** $\exists O_i \in (ChemainCritique \cap \Pi'(FPGA_j))\forall(i,j)$ **then**
6: $S_{SG} = \{sous - graphe \in \Pi'(FPGA_j)\forall j\}$
7: **for all** $SG \in S_{SG}$ **do**
8: Heuristique SynDEx-IC $(SG, C = 0)$
9: **end for**
10: $G''_{AR} = \text{durations\_update}(G'_{AR})$
11: Heuristique SynDEx (G_{AL}, G''_{AR})
12: **for all** $SG \in S_{SG}$ **do**
13: Heuristique SynDEx-IC $(SG, C = flex(SG) + d\_min(SG))$
14: **end for**
15: **end if**

En résumé, l'heuristique de distribution/ordonnancement de SynDEx (étape de 1 à 3) est suivi d'une heuristique d'optimisation de la partie reconfigurable.

Nous allons maintenant revoir en détail ces deux parties à travers un exemple complet de prise de décision de l'intra prédiction 16x16 de la norme de compression H.264.

L'intra prédiction 16x16 consiste à prédire le macro-bloc de pixels à partir des pixels voisins suivant différents modes. La norme H.264 spécifie 4 modes de prédictions : le mode vertical, le mode horizontal, le mode DC et le mode "plane". Une étude présentée dans [Kessentini et al., 2008] justifie l'élimination du mode "plane" car c'est un mode de prédiction très complexe qui apporte une amélioration minime de la qualité de la prédiction. Ainsi, on ne considère que les modes de prédiction vertical, horizontal et DC.

Le mode de prédiction donnant le macro-bloc prédit le plus ressemblant au macro-bloc courant (à prédire) est choisi. Le critère de comparaison utilisé est le **Sum of Absolute Difference** (SAD) donné par l'équation 4.5 (avec MBcour : macro-bloc courant, MBref : macro-bloc de référence et N : taille du macro-bloc). Ce critère de comparaison consiste en une répétition de 256 différences en valeur absolue et accumulations.

$$SAD = \sum_{i=1}^{N} \sum_{j=1}^{N} |MBcour(i,j) - MBref(i,j)| \tag{4.5}$$

L'algorithme de décision de mode intra 16x16 a trois entrées : 16 pixels de voisinage haut, 16 pixels de voisinage gauche et le macro-bloc source de 16x16 pixels. Les résultats de cet algorithme sont la valeur moyenne des pixels voisins (haut et gauche), le mode de prédiction qui est un entier valant 0, 1 ou 2 et le SAD du meilleur macro-bloc prédit par rapport au macro-bloc source. L'algorithme commence par calculer la valeur moyenne des pixels de voisinage, calcule les SAD de chacun des macro-blocs prédits selon le mode DC, le mode vertical et le mode horizontal, puis compare les trois SAD trouvés et donne le minimum des SAD et le mode de prédiction lui correspondant.

L'architecture qu'on cible est composée par un processeur et un FPGA connectés à travers leurs ports ethernet.

4.8.1 Heuristique de Distribution-Ordonnancement

Cette première partie se compose de trois étapes :

* Spécification de l'algorithme et de l'architecture.
* Transformation du graphe d'architecture.
* partitionnement/ordonnancement.

Spécification de l'architecture et de l'algorithme

Cette étape correspond aux deux premières lignes de l'algorithme 3 et elle vise à définir les graphes d'algorithme et d'architecture. Ainsi, le graphe d'architecture est défini par un ensemble d'opérateurs programmables, un ensemble d'opérateurs reconfigurables et les différents moyens de communications connectant tous ces opérateurs.

Comme SynDEx est utilisé pour la spécification de ces graphes d'architecture et d'algorithme, cette spécification utilise les modèles de AAA pour les composants programmables (non étendus). Ce modèle d'architecture ne fait pas la différence entre composants programmables et composants reconfigurables, donc pour différencier ces deux types d'opéra-

teurs, l'utilisateur représente chaque opérateur reconfigurable dans le graphe d'algorithme par un opérateur dont le nom commence par « fpga ».

L'interface de SynDEx ne permet que la spécification de la durée d'exécution des opérations sur chaque opérateur capable de l'exécuter. Pourtant l'optimisation de l'implémentation des opérations sur les composants reconfigurables (par SynDEx-IC) nécessite de connaitre la surface occupée par chaque opération. Pour ne pas avoir à modifier l'interface de SynDEx, cette surface occupée par chaque opération est spécifiée dans un fichier texte de la façon suivante : chaque ligne contient la spécification d'une opération sous la forme "nom_opération = surface_occupée".

Une autre contrainte de spécification doit être respectée par l'utilisateur : les opérations répétitives qui peuvent s'exécuter sur les composants reconfigurables, doivent être encapsulées dans une autre opération pour pouvoir optimiser leurs implémentations. En effet, SynDEx défactorise complètement et systématiquement chaque frontière de factorisation du graphe d'algorithme et considère chaque itération comme opération indépendante. Ainsi, il peut distribuer les différentes itérations d'une seule frontière de factorisation sur des opérateurs différents. En encapsulant les opérations répétitives dans une autre opération, SynDEx les considère comme une seule opération et ne les défactorise pas. On peut alors faire appel à SynDEx-IC pour optimiser l'implémentation des frontières de factorisation distribuées sur les composants reconfigurables.

Le graphe d'algorithme de prise de décision de l'intra 16x16 de la norme H.264 est représenté par la figure 4.19. Ce graphe d'algorithme comporte 3 capteurs (sensor) pour ces entrées : TOP pour la ligne de voisinage haut de 16 pixels, LEFT pour la colonne de voisinage gauche de 16 pixels et SRC le MB source de 16x16 pixels. Les actionneurs de ce graphe d'algorithme sont DC_Val pour la valeur moyenne des pixels voisinages, best_mode pour le mode de prédiction choisi et min_SAD pour le SAD du meilleur MB prédit. Le graphe d'algorithme comporte aussi 5 fonctions : cal_DC_val pour calculer la valeur moyenne des pixels de voisinage, SAD_DC, SAD_V et SAD_H pour calculer les SAD respectifs des modes DC, vertical et horizontal et la fonction comparateur_SAD pour comparer les trois SAD trouvés puis donner le minimum des SAD et le mode de prédiction lui correspondant. Comme nous l'avons vu dans l'équation du SAD (équation 4.5), elle consiste en une répétition de 256 différences en valeur absolue et accumulations. Ces répétitions sont encapsulées dans les blocs SAD_DC, SAD_V et SAD_H.

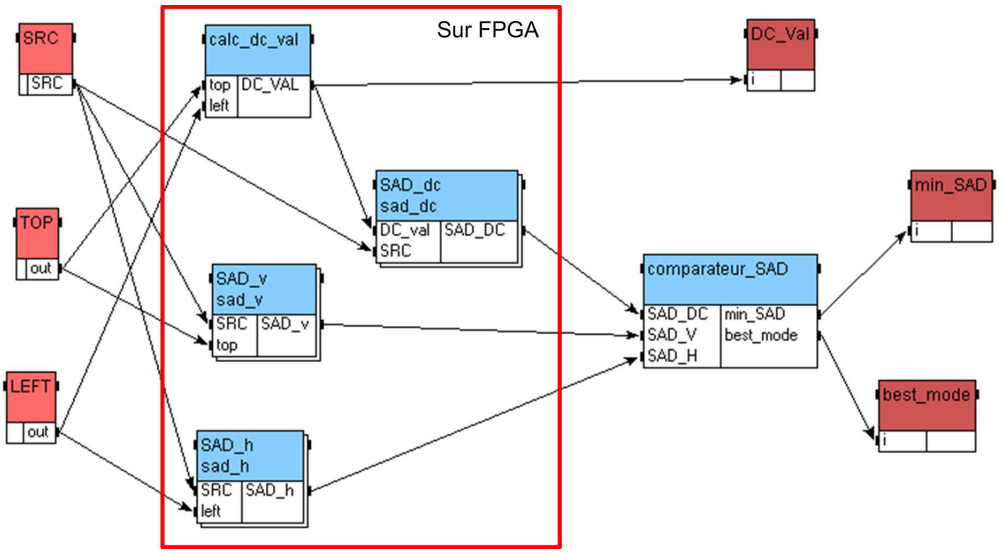

FIGURE 4.19 – graphe d'algorithme de prise de décision de l'intra 16x16

L'architecture ciblée comporte deux opérateurs : un processeur et un FPGA. Donc le graphe d'architecture qu'on utilise est composé par un opérateur programmable (CPU) et un composant reconfigurable (FPGA1). Comme précisé à la page 81, à ce stade du flot de conception, il n'y a pas de différences entre les deux opérateurs utilisés. Afin de les différencier, le nom de l'opérateur qui modélise le composant reconfigurable commence alors par FPGA. La liaison ethernet connectant ces deux opérateurs est modélisée par une SAM point à point (FPGA_LINK). Le graphe d'architecture obtenue est représenté par la figure 4.20.

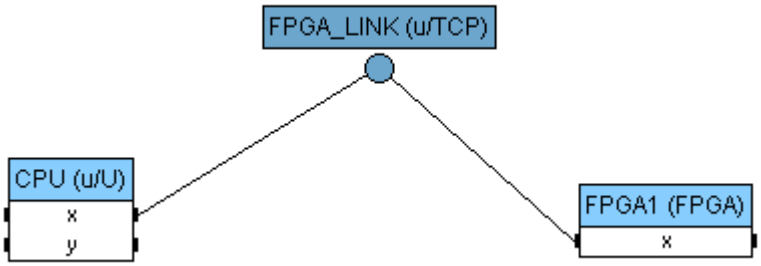

FIGURE 4.20 – graphe d'architecture

Les durées d'exécution en cycles des opérations de l'algorithme sur les deux opérateurs de l'architecture sont regroupées dans le tableau 4.3.

Tableau 4.3 – Durées d'exécution des opérations sur les différents opérateurs

Opération	CPU	FPGA1
SRC	256	
TOP	16	
LEFT	16	
calc_DC_val	33	18
SAD_V	768	296
SAD_H	768	296
SAD_DC	768	296
comparateur_SAD	38	
DC_Val	1	
min_SAD	1	
best_mode	1	

On remarque que les durées d'exécution sur le composant reconfigurable ne sont données que pour les opérations calc_DC_val, SAD_V, SAD_H et SAD_DC. C'est un choix fait pour spécifier que seulement ces opérations peuvent être distribuées sur le composant reconfigurable FPGA1 (nous les avons encadrés d'un rectangle pour les mettre en évidence sur la figure 4.19).

Transformation du graphe d'architecture

En exécutant l'heuristique de SynDEx sur les graphes d'architecture (figure 4.20) et d'algorithme (figure 4.19), nous obtenons l'implémentation présentée dans le tableau 4.4.

Ce tableau associe à chaque opération du graphe d'algorithme l'opérateur qui l'exécute et ses dates de début et de fin. Dans cette solution, on remarque que les seuls opérations distribuées sur FPGA sont SAD_DC et SAD_V et ces deux opérations sont ordonnancées en série (la date de début de l'une est la date de fin de l'autre), alors qu'elles peuvent être exécutées en parallèle sur le FPGA. Cette solution montre comme expliqué en 4.6 que SynDEx n'est pas adapté à faire la distribution des algorithmes sur des architectures contenant des FPGA puisque le parallélisme interne est caché à SynDEx. La longueur du chemin critique de cette implémentation est 1211.

Tableau 4.4 – Résultats de distribution ordonnancement avant transformation de graphe.

Opération	opérateur	Date de début	Date de fin
SRC	CPU	0	256
TOP	CPU	256	272
LEFT	CPU	272	288
calc_DC_val	CPU	288	321
DC_Val	CPU	321	322
SAD_H	CPU	322	1090
SAD_DC	FPGA	578	874
SAD_V	FPGA	874	1170
comparateur_SAD	CPU	1171	1209
min_SAD	CPU	1209	1210
best_mode	CPU	1210	1211

Il est donc nécessaire de transformer le graphe d'architecture spécifié par l'utilisateur vers un graphe utilisant notre extension du modèle d'architecture.

Cette transformation (ligne 3 de l'algorithme 3) s'effectue automatiquement selon l'algorithme 4

Algorithm 4 Algorithme de transformation du graphe d'architecture

1: $G'_{AR} = G_{AR}$
2: **for all** $FPGA_i \in G_{AR}$ **do**
3: $FPGA'_i = \{\}$
4: **for all** $port_j \in FPGA_i$ **do**
5: $FPGA'_i = FPGA'_i + \{COM_j\}$
6: **end for**
7: **for all** $O_k \in \lambda^{-1}(FPGA_i)$ **do**
8: $FPGA'_i = FPGA'_i + \{Oprd_k\}$
9: **end for**
10: $FPGA'_i = FPGA'_i + \{med_{inter-Oprd}\}$
11: $G'_{AR} = G'_{AR} - \{FPGA_i\} + \{FPGA'_i\}$
12: **end for**

Cette transformation consiste à remplacer chaque composant reconfigurable par l'ensemble d'opérateurs qu'il pourrait contenir. Ainsi chaque opérateur du graphe d'algorithme dont le nom commence par FPGA est remplacé par :

* Pour chaque port un opérateur de communication (lignes 4-6 de l'algorithme 4) : c'est un opérateur qui n'exécute aucune opération, néanmoins il est essentiel pour modéliser l'aspect séquentiel des communications.
* Pour chaque opération pouvant s'exécuter sur le composant reconfigurable en question un opérateur dégénéré (lignes 7-9 de l'algorithme 4) : ce type d'opérateur (défini dans la section 4.2.5) n'est capable d'exécuter qu'un seul type d'opération du graphe d'algorithme. L'utilisation d'un opérateur dégénéré pour chaque opération exécutable sur le composant reconfigurable permet de modéliser le parallélisme potentiel offert par ces composants. Ainsi toutes les opérations distribuées sur le composant reconfigurable peuvent s'exécuter indépendamment les unes des autres.
* Un moyen de connexion (ligne 10 de l'algorithme 4) : c'est une SAM multipoint permettant la diffusion matérielle des données, utilisée pour connecter les opérateurs dégénérés (de traitement et de communication) entre eux. Conformément au modèle de communication exposé à la section 4.3, le temps de traversé de cette SAM est nul pour tout type de données.

Ce graphe d'architecture remplace le graphe d'architecture initial dans le fichier du projet de SynDEx. La caractérisation temporelle spécifiée au début doit être modifiée pour l'adapter à cette transformation du graphe d'architecture car on n'a plus d'opérateur FPGA. On associe la durée d'exécution des opérations initialement définies pour le FPGA à l'opérateur capable de l'exécuter contenu dans ce FPGA.

Pour notre exemple de l'algorithme de prise de décision de l'intra 16x16, le graphe d'architecture spécifié contient un seul opérateur dont le nom commence par FPGA. Cet opérateur représente un composant reconfigurable. Il est capable d'exécuter 4 opérations : calc_DC_val, SAD_V, SAD_H et SAD_DC (tableau 4.3). Cet opérateur possède un seul port. Donc la transformation automatique du graphe le remplace par un opérateur de

communication et 4 opérateurs dégénérés. La figure 4.21 représente le graphe d'architecture modifié. Dans cette figure, les opérateurs qui remplacent le composant reconfigurable sont encadrés.

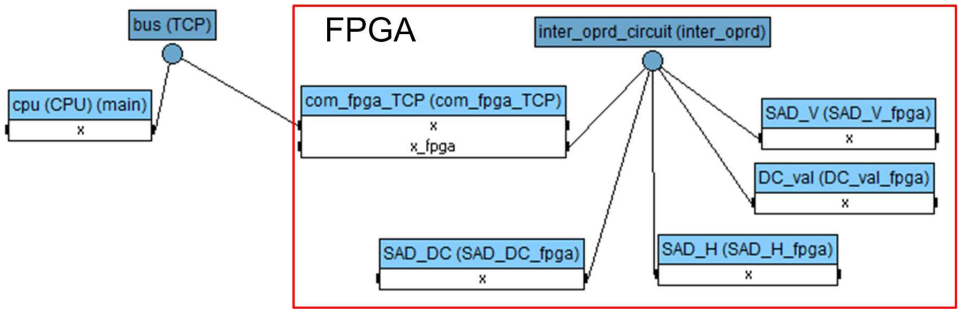

FIGURE 4.21 – Graphe d'architecture modifié

Partitionnement/Ordonnancement

L'étape suivante consiste à effectuer le partitionnement/ordonnancement des différentes opérations de l'algorithme sur les différents composants qui constituent l'architecture. L'heuristique de SynDEx permet cette opération. En effet, l'heuristique de SynDEx est exécutée en mode "no flatten" (on n'y tient compte que du plus haut niveau hiérarchique de l'algorithme) pour ne pas défactoriser systématiquement les frontières de factorisation encapsulées dans une autre opération (section 4.8.1). On utilise le graphe d'algorithme spécifié par l'utilisateur et le graphe d'architecture modifié obtenu à la fin de l'étape précédente (section 4.8.1). On obtient alors pour chaque opération du graphe d'algorithme la date de début et de fin et le nom de l'opérateur sur lequel elle est distribuée.

Pour l'exemple de l'algorithme de prise de décision de l'intra 16x16 de la norme H.264, l'exécution de l'heuristique de SynDEx pour le partitionnement/ordonnancement des opérations du graphe d'algorithme de la figure 4.19 sur les opérateurs du graphe d'architecture transformé de la figure 4.21 donne les résultats présentés dans le tableau 4.5.

Tableau 4.5 – Résultats de distribution ordonnancement après transformation de graphe.

Opération	opérateur	Date de début	Date de fin
SRC	CPU	0	256
TOP	CPU	256	272
LEFT	CPU	272	288
calc_DC_val	CPU	288	321
DC_Val	CPU	321	322
SAD_H	FPGA	578	874
SAD_DC	FPGA	578	874
SAD_V	FPGA	578	874
comparateur_SAD	CPU	877	915
min_SAD	CPU	915	916
best_mode	CPU	916	917

Pour des raisons de simplification du tableau, on ne présente pas le nom de l'opérateur dégénéré qui exécute les opérations distribuées sur le composant reconfigurable, mais il

est implicitement connu car chaque opérateur dégénéré de ce composant ne peut exécuter qu'une seule opération. Dans cette solution, les trois opérations de calcul de SAD sont distribuées sur le FPGA et s'exécutent en parallèle (ils ont les mêmes dates de début et de fin). Comme prévu la transformation de graphe effectuée (4.8.1) a permis d'exposer le parallélisme potentiel du FPGA pour que SynDEx puisse en tenir compte. La longueur du chemin critique passe de 1211 à 917 grâce à l'exploitation du parallélisme du FPGA.

4.8.2 Optimisation FPGA

Dans l'étape précédente, le graphe d'algorithme a été distribué/ordonnancé sur les parties programmables et reconfigurables du graphe d'architecture. Dans cette étape, il s'agit d'essayer d'optimiser les sous-graphes distribués sur les composants reconfigurables.

On définit un sous-graphe par un ensemble connexe d'opérations du graphe d'algorithme et un sous-graphe sur FPGA par un sous-graphe dont toutes les opérations sont distribuées sur FPGA.

L'optimisation de l'implémentation que nous effectuons a pour but de réduire la latence totale de l'algorithme tout en utilisant le plus petit nombre de blocs logiques. Pour réduire la latence totale de l'algorithme, il faut réduire le temps d'exécution des opérations situées sur le chemin critique. Il faut pour cela s'attaquer à l'optimisation de l'implémentation des opérations appartenant au chemin critique et distribuées sur un composant programmable. En effet, comme nous avons vu en section 3.4.4, on peut réduire le temps d'exécution des opérations sur les composants reconfigurables en augmentant le niveau de parallélisation si la dépendance de données le permet. Si la distribution donnée par SynDEx ne contient pas d'opération distribuée sur composant reconfigurable et appartenant au chemin critique, on ne peut pas encore optimiser l'implantation. Dans ce cas, on passe alors à la génération des exécutifs qui sera présentée dans le chapitre 5.

SynDEx-IC est utilisé pour optimiser l'implémentation des opérations distribuées sur FPGA. Cette optimisation se déroule en deux étapes :

1. Défactorisation complète de toute les frontières de factorisation. Ainsi, nous obtenons une implémentation avec une latence minimale et un nombre de blocs logiques maximal (figure 3.14).
2. Pour réduire l'utilisation des blocs logiques sans augmentation de la latence de l'algorithme, la refactorisation de ces frontières de factorisation est effectuée tant qu'ils ne sont pas sur le chemin critique.

Le chemin critique de l'implémentation proposée par SynDEx pour l'algorithme de prise de décision de l'intra 16x16 (tableau 4.5), est constitué par les opérations : SRC, TOP, LEFT, SAD_V, comparateur_SAD min_SAD et best_mode. Parmi ces opérations, l'opération SAD_V est distribuée sur le composant reconfigurable. Il semble donc possible d'optimiser la latence totale de cette implémentation en défactorisant les frontières de factorisation distribuées sur ce composant reconfigurable, à savoir les opérations SAD_V, SAD_H et SAD_DC.

Principe de l'optimisation temporelle

Nous visons toujours la réduction du temps d'exécution de l'algorithme, c'est à dire la longueur du chemin critique. Nous devons pour atteindre ce but exploiter au maximum le parallélisme potentiel de l'algorithme. Ce parallélisme est partiellement exploité par SynDEx lors du partitionnement/ordonnancement. Mais les algorithmes renferment en

général des répétitions dont les itérations peuvent être indépendantes, donc fortement parallélisable. Ce fort parallélisme est exploitable grâce à l'utilisation des composants reconfigurables.

Pour atteindre une implémentation avec une latence minimale, nous défactorisons totalement toutes les frontières de factorisation distribuées sur les composants reconfigurables. Pour ce faire, on exécute l'heuristique SynDEx-IC sur chaque sous-graphe sur FPGA en imposant une contrainte temporelle nulle (lignes 7-9 de l'algorithme 3). Cela a bien pour effet de construire une implémentation avec un temps d'exécution minimale mais qui utilise un nombre maximal de blocs logiques et par conséquent un coût élevé. Pour essayer de réduire le coût de cette implémentation, il faut passer par une "optimisation de la surface occupée".

Nous avons vu que chaque opération de calcul de SAD dans l'implémentation proposée par SynDEx pour l'exemple choisi, constitue un sous-graphe sur FPGA. Nous utilisons alors SynDEx-IC pour optimiser l'implémentation de ces trois sous-graphes en défactorisant complètement les frontières de factorisation qu'ils contiennent. Ainsi l'heuristique de SynDEx-IC est exécutée trois fois en lui passant à chaque fois une des opérations de SAD comme graphe d'algorithme. Rappelons que l'opération SAD est constituée par une somme de 256 différences en valeur absolue. Toutes ces différences en valeur absolue sont indépendantes et peuvent s'effectuer en parallèle. La nouvelle durée d'exécution obtenue pour chacune des opération SAD après défactorisation totale est 41. En utilisant cette durée, SynDEx donne les dates du tableau 4.6.

Tableau 4.6 – Résultats de distribution ordonnancement après optimisation temporelle.

Opération	opérateur	Date de début	Date de fin
SRC	CPU	0	256
TOP	CPU	256	272
LEFT	CPU	272	288
calc_DC_val	CPU	288	321
DC_Val	CPU	321	322
SAD_H	FPGA	578	619
SAD_DC	FPGA	578	619
SAD_V	FPGA	578	619
comparateur_SAD	CPU	622	660
min_SAD	CPU	660	661
best_mode	CPU	661	662

La longueur du chemin critique est réduite de 917 à 662 grâce à la réduction de la durée d'exécution des opérations de SAD.

Principe de l'optimisation de la surface occupée

Après l'optimisation temporelle, nous avons obtenu une implémentation avec un temps d'exécution minimal (si on ne remet pas en cause la distribution donnée par SynDEx) mais utilisant un nombre maximal de blocs logiques. Alors que tout concepteur des systèmes embarqués est soumis à des contraintes de coût. Nous devons donc réduire le nombre de ressources logiques utilisées sans augmenter la latence de l'algorithme. La réduction du temps d'exécution des opérations ne se trouvant pas sur chemin critique, ne réduit pas la latence totale de l'algorithme mais augmente la surface occupée donc son coût. Pour ces

raisons, nous devons refactoriser les frontières de factorisation qui n'appartiennent pas au chemin critique. En d'autres termes, on refactorise les frontières de factorisations des sous-graphes sur FPGA optimisé précedemment tant que leur flexibilité d'ordonnancement n'est pas nulle. On rappelle que la flexibilité d'ordonnancement d'une opération (ou un sous-graphe) est la différence entre la plus grande date de début de cet opération (ou sous-graphe) n'engendrant pas de rallongement du chemin critique et la plus petite date de début possible (page 46).

Rappelons aussi que l'heuristique de SynDEx-IC a pour but de trouver une implémentation qui satisfait une contrainte temporelle en utilisant le minimum possible de blocs logiques. Ainsi, pour pouvoir réduire au maximum le nombre de blocs logiques utilisés, on utilise SynDEx-IC en lui passant comme contrainte temporelle la somme de la durée minimale d'exécution (trouvée après l'optimisation temporelle) et la flexibilité du sous-graphe. Cette étape est décrite par les lignes 12-14 de l'algorithme 3. Cette étape nécessite en aval que SynDEx recalcule des dates de début et de fin des opérations en tenant compte des résultats de l'optimisation temporelle (lignes 10 et 11 de l'algorithme 3).

Après l'optimisation temporelle de l'exemple étudié, l'opération SAD_V est encore sur le chemin critique donc ne peut pas être refactoriser sans allonger le chemin critique. Par contre les opérations SAD_H et SAD_DC ne sont pas sur le chemin critique et leurs flexibilités d'ordonnancement sont respectivement 1 et 2. La contrainte temporelle utilisée pour refactoriser ces deux opérations sont $41 + 1 = 42$ pour SAD_H et $41 + 2 = 43$ pour SAD_DC. Après cette dernière étape d'optimisation, on obtient les dates présentées dans le tableau 4.7.

Tableau 4.7 – Résultats finals de distribution ordonnancement.

Opération	opérateur	Date de début	Date de fin
SRC	CPU	0	256
TOP	CPU	256	272
LEFT	CPU	272	288
calc_DC_val	CPU	288	321
DC_Val	CPU	321	322
SAD_H	FPGA	578	619
SAD_DC	FPGA	578	620
SAD_V	FPGA	578	620
comparateur_SAD	CPU	622	660
min_SAD	CPU	660	661
best_mode	CPU	661	662

On remarque que les durées d'exécution des opérations SAD_DC et SAD_V ont augmenté (de 41 à 42). Cette augmentation est due à une refactorisation partielle de ces deux opérations. Malgré cette refactorisation, la longueur du chemin critique reste inchangée (662) car l'augmentation des durées d'exécution de ces opérations ne dépassent pas leurs flexibilités respectives.

4.9 Evaluation des performances

Le tableau 4.8 regroupe les caractéristiques des méthodes de partitionnement matériel/logiciel présentées. La plus part des méthodes présentées (ou rencontrées dans la

Tableau 4.8 – Tableau comparatif des méthodes de partitionnement

Méthode	Architecture cible	exploration matérielle	partitionnement
ECOS	processeur + coprocesseurs	non	manuel + automatique
Eles et al.	processeur + coprocesseurs	non	automatique guidé
Zaho et al.	processeur + coprocesseurs	non	automatique
Han et al.	multiprocesseur + coprocesseur	non	automatique
Srinivsan et al.	processeur + coprocesseurs	oui	automatique
algo proposé	multiprocesseur + coprocesseurs	oui	manuel + automatique

littérature) ciblent une architecture composée par un processeur et un coprocesseur. De ces cinq méthodes présentées, seule la méthode Han et al. [Han et al., 2013] (comme la notre) effectue le partitionnement matériel/logiciel en ciblant une architecture multiprocesseur avec coprocesseur. Cette méthode effectue le partitionnement ordonnancement des opérations sur une architecture multiprocesseur (ne considère pas les composants matériels) puis une optimisation temporelle est effectuée en choisissant des opérations à implémenter en matériel. Ce partitionnement en deux étapes séparées peut mener à une sous optimisation de la solution car la première étape de partitionnement ne considère pas l'effet de la seconde étape (le processeur devient libre si l'opération est transférée en matériel). Par contre dans notre méthode, le partitionnement considère tous les opérateurs (programmables ou reconfigurables).

Peu d'algorithmes permettent l'exploration automatique de l'espace d'implémentations matérielles. L'algorithme que nous proposons utilise l'heuristique de SynDEx-IC pour explorer les différents degrés de déroulement de boucles contenues dans les parties implémentées en matériel. Cette exploration de l'espace d'implémentation s'effectue après l'étape de partitionnement ordonnancement. En conséquence, le partitionnement matériel/logiciel ne tient compte que d'une seule implémentation matérielle possible (la pire en terme de durée d'exécution). Par contre, la méthode proposée par Srinivsan et al. [Srinivasan et al., 1998] considère les différentes implémentations matérielles possibles lors du partitionnement matériel/logiciel. Donc l'espace de solutions total est plus grand et par la suite la solution choisie peut éventuellement être meilleure que celle résultant d'un espace de solutions réduit. Cette amélioration éventuelle de la solution choisie s'accompagne d'une augmentation du temps de recherche de cette solution.

Plusieurs des méthodes de partitionnement matériel/logiciel automatique, comme notre algorithme, donnent la possibilités à l'utilisateur de guider le choix vers l'implémentation de quelques opérations en matériel ou en logiciel ou d'imposer ce choix. Pour l'algorithme d'Eles et al. [Eles et al., 1994a], le concepteur du système peut définir les poids de multiplication de la fonction coût utilisée et par la suite guider l'algorithme vers optimiser un aspect de la solution plus que d'autres. Notre algorithme et le project ECOS [Freund et al., 1997] permettent d'imposer l'implémentation matérielle ou logicielle de quelques (ou toutes) opérations de l'algorithme. Le choix du type d'implémentation pour le reste des opérations de l'algorithme se fait automatiquement.

L'algorithme de couplage que nous proposons se décompose en six étapes successives : la transformation de graphe, le partitionnement/ordonnancement, le test d'optimisation temporelle, l'optimisation temporelle, le calcul des nouvelles dates et l'optimisation de surface. Considérons un graphe d'algorithme contenant q opérations et un graphe d'architecture comportant n composants programmables et m composants reconfigurables. La

transformation de graphe a alors une complexité linéaire en $O(m)$. Notons la complexité de l'heuristique de SynDEx pour un graphe d'algorithme de d opérations et un graphe d'architecture de e opérateurs par $C\_SynDEx(d, e)$. Supposons que la transformation du graphe crée k opérateurs dégénérés pour remplacer les composants reconfigurables de l'architecture. La complexité de l'étape de partitionnement est $C\_SynDEx(q, n + k)$. Le test de possibilité d'optimisation a une complexité linéaire en $O(q)$. Notons la complexité de l'heuristique de SynDEx-IC pour un graphe d'algorithme de d opérations par $C\_SynDEx - IC(d)$. Supposons que la distribution donnée par SynDEx donne p sous-graphes distribués sur FPGA contenant qp_i opérations chacun, la complexité de l'optimisation temporelle est $p \times C\_SynDEx - IC(qp_i)$. Puisque le calcul des nouvelles dates s'effectue en exécutant SynDEx en contraignant la distribution, la complexité de cette étape s'exprime par $C\_SynDEx(q, 1)$. Enfin la complexité de l'optimisation de surface s'exprime par $p \times C\_SynDEx - IC(qp_i)$. Puisque l'algorithme de couplage que nous proposons est formé par la succession de ces étapes, la complexité totale de cet algorithme est la somme des complexités de chacune de ces étapes.

4.10 Conclusion

Une extension du modèle d'architecture AAA a été présentée dans ce chapitre. Cette extension permet de modéliser les composants reconfigurables. Après un algorithme de couplage des outils SynDEx/SynDEx-IC, est présenté pour permettre le partitionnement automatique des algorithmes sur les plate-formes mixtes et l'optimisation de leurs implémentations.

Après l'optimisation de l'implémentation, nous pouvons passer à la génération automatique des codes correspondants. Le chapitre suivant exposera la méthode de génération du code de l'application et détaillera le circuit permettant de gérer la communication et la synchronisation des opérateurs contenus dans les FPGA avec les autres composants qui constituent l'architecture.

Les travaux décrits dans ce chapitre sont publiés dans [Feki et al., 2013]
et [Feki et al., 2014b].

Chapitre 5

Génération automatique des exécutifs

5.1 Introduction

Au cours des chapitres précédents, nous avons étendu les modèles AAA pour pouvoir modéliser un algorithme, une architecture mixte et l'implémentation. Nous avons ensuite présenté et évalué une heuristique d'optimisation de l'implémentation sur les architectures mixtes. Dans ce chapitre, nous allons étendre les techniques de génération de code AAA afin de générer l'exécutif complet d'une application sur une architecture mixte. Cela comprend le code compilable côté programmable, le VHDL synthétisable côté reconfigurable mais aussi le code des communications entre les composants programmables et reconfigurables.

SynDEx, étant conçu pour les architectures multi-composant peut générer les macros de communication et synchronisation. Par contre SynDEx-IC, étant conçu pour les mono-FPGA, ne génère pas le code VHDL de composants qui gère la communication et la synchronisation du FPGA avec les autres composants de l'architecture. Nous allons dans ce chapitre présenter la façon selon laquelle SynDEx et SynDEx-IC sont utilisés pour générer l'exécutive de l'application sur l'architecture mixte. Ensuite, nous présentons une IP générique pour gérer les communications et synchronisation du FPGA avec les autres composants.

5.2 Rappel

Comme présenté dans le chapitre 3, la méthodologie AAA permet après optimisation de l'implémentation de générer automatiquement des exécutifs pour chaque opérateur sous la forme de macro code générique. En effet, ces exécutifs sont génériques et ne dépendent donc pas du langage cible de chaque opérateur. Ensuite un macro-processeur et des librairies d'exécutifs sont utilisés pour traduire ces exécutifs génériques en exécutifs compilables par le compilateur de l'opérateur cible.

5.2.1 Génération des exécutifs par SynDEx

Les exécutifs générés par SynDEx fournissent des services permettant la bonne exécution de l'application. Cela comprend le code de l'application, l'allocation de mémoires, les communications et les synchronisations, etc .... Ces exécutifs sont générés sous forme de macro-code et d'une bibliothèque générique et peuvent être facilement traduits en langage spécifique à tout processeur grâce à un macro-processeur et des bibliothèques spécifiques

aux cibles. Ces exécutifs sont parfaitement adaptés à l'application pour la quelle ils sont générés : on dit qu'ils sont taillés sur mesure. Comme ces exécutifs remplacent le système d'exploitation temps réel (RTOS), on peut garantir l'aspect déterministe de l'exécution. La génération d'exécutifs se fait en trois étapes [Grandpierre and Sorel, 2003] :

* Transformation de l'implémentation optimisée en graphe d'exécution : cette transformation s'effectue en ajoutant de nouveaux types de sommets (*Loop*, *EndLoop*, *pre-ful/suc-full*, *pre-empty/suc-empty*). Comme SynDEx vise l'optimisation de l'implémentation d'applications réactives (constamment en interaction avec leur environnement), il faut rendre la séquence d'opération attribuée à chaque opérateur répétitive. Donc ces séquences d'opérations sont encadrées par des sommets *Loop* au début et *EndLoop* à la fin. Les autres sommets ajoutés concernent la synchronisation entre les différents opérateurs et communicateurs qui exécutent des opérations avec dépendances de données. Afin de réaliser cette synchronisation on ajoute, après chaque opération productrice d'une donnée dont dépend l'exécution d'une autre opération sur un autre opérateur (ou communicateur), un sommet *pre-full* et avant l'opération consommatrice un sommet *suc-full*. Pour que le contenu du registre ne soit pas modifié avant son utilisation, on ajoute avant l'opération productrice un sommet *suc-empty* et après l'opération consommatrice un sommet *pre-empty*. Ces sommets *pre-full*, *suc-full*, *pre-empty* et *suc-empty* modélisent des fonctions capable de lire et modifier des sémaphores binaires.
* Transformation du graphe d'exécution en séquences de macro-exécutif : chaque sous-graphe du graphe d'exécution associé à un opérateur est traduit en un fichier de macro-exécutif. Le passage par ces macros intermédiaires permet de cibler plusieurs types de langages de programmation comme nous verrons plus loin. Ces fichiers sont constitués de trois parties : l'allocation de mémoire, les séquences de communication et la séquence d'opérations de calcul.

  La liste de macros d'allocation de mémoire contient une macro *alloc_ (type, nom, taille)* pour chaque sommet allocation distribué sur une RAM connectée à l'opérateur. *Type* est le type de la donnée, *nom* est le nom de l'opération productrice et du port correspondant et *taille* représente la taille du vecteur.

  Les fichiers de macros contiennent une séquence de communication pour chaque communicateur connecté au opérateur. Cette séquence est générée entre les macros *com_ thread_* et *end_ thread_* . Ces séquences de macros sont générées en respectant l'ordre des sommets associés au communicateur correspondant.

  La séquence de macros de calcul est générée entre les macros *main_* et *end_ main_* . Chaque sommet opération est traduit en macro tout en respectant l'ordre d'exécution.
* Traduction des macro-exécutifs en exécutifs compilables : chaque séquence de macros est traduite en code source dans le langage cible de l'opérateur en utilisant un macro-processeur (comme gnu M4).

5.2.2 Génération des exécutifs par SynDEx-IC

La génération automatique de code de SynDEx-IC vise à fournir un code compatible avec les outils de synthèse et de simulation et qui supporte facilement de nouveaux langages de description matérielle à partir d'une spécification haut niveau. Cette génération de code s'effectue en deux étapes :

* Génération de macro-code générique : c'est une étape intermédiaire indépendante du choix de langage de description matérielle choisi. Elle permet ainsi une meilleur portabilité du code. Les macros générés sont de deux types : les macros de déclaration et de définitions et les macros d'interconnexion des composants. Chaque fichier de macro-code généré comporte : des macros d'inclusion de bibliothèques, des macros de définitions de l'ensemble des composants, des macros de définitions des signaux et des macros de définition des interconnexions entre les composants et les signaux.
* Traduction du macro-code en code RTL synthétisable : cette traduction s'effectue en utilisant le macro-processeur m4 et une ou plusieurs bibliothèques contenant les définitions des macros.

5.3 Techniques de communication entre composants reconfigurables et composants programmables

5.3.1 Communication par mémoire partagée

Le transfert de données entre la partie matérielle et la partie logicielle peut s'effectuer en utilisant une mémoire partagée. L'émetteur sauvegarde la donnée à transmettre dans la mémoire partagée à une adresse connue par le récepteur. Puis, ce dernier lit ces données directement de la mémoire partagée. Cette méthode nécessite une connaissance préalable des données à transférer et une bonne synchronisation entre les deux composants. Cette méthode peut en effet mener au problème de cohérence de cache. Ce problème se pose pour les architectures multicomposants de haute performance. En effet, les processeurs pour ces architectures sont munis de mémoires caches inaccessibles pour les autres composants du système. Lorsque un processeur P utilise une variable de la mémoire, il en garde une copie dans sa cache pour les éventuelles utilisations ultérieures. Si la valeur de cette variable est modifiée par un autre composant, ce processeur P utilisera pour ces prochains calculs la valeur de sa cache qui est obsolète. Ce problème peut être évité en utilisant les instructions spécifiques d'invalidation de la cache.

La figure 5.1 représente un schéma simplifié d'une architecture utilisant une mémoire double port pour la communication entre deux processeurs ou FPGA. Ce type de mémoire permet à chaque processeur ou FPGA d'y accéder à travers un port d'adresses, un port de données et un port de contrôle qui lui sont dédiés. Dans ce cas, les deux processeurs ou FPGA peuvent accéder à la mémoire simultanément et il n'est pas nécessaire d'utiliser un arbitre pour gérer l'accès à la mémoire.

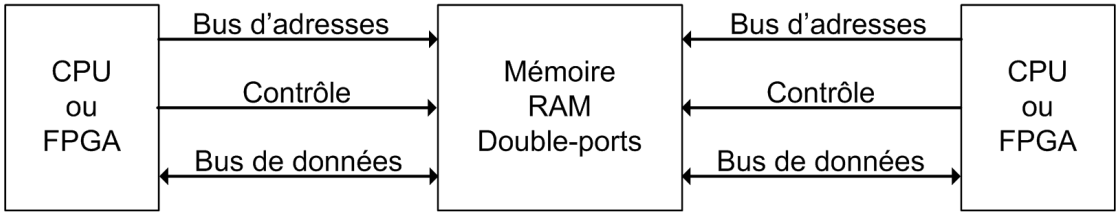

FIGURE 5.1 – Communication par mémoire double port

La figure 5.2 représente une architecture comportant deux processeurs ou FPGA connectés à travers une mémoire simple port partagée. Contrairement à l'utilisation des

mémoires double ports, l'accès de tous les processeurs et FPGA à la mémoire partagée s'effectue à travers un seul port pour chaque type d'informations (données, adresses et contrôle). Donc un système d'arbitrage/multiplexage est nécessaire pour gérer l'accès à la mémoire. Une partie ou tout l'ensemble mémoire/arbitre/multiplexeur peut être intégrée dans un FPGA.

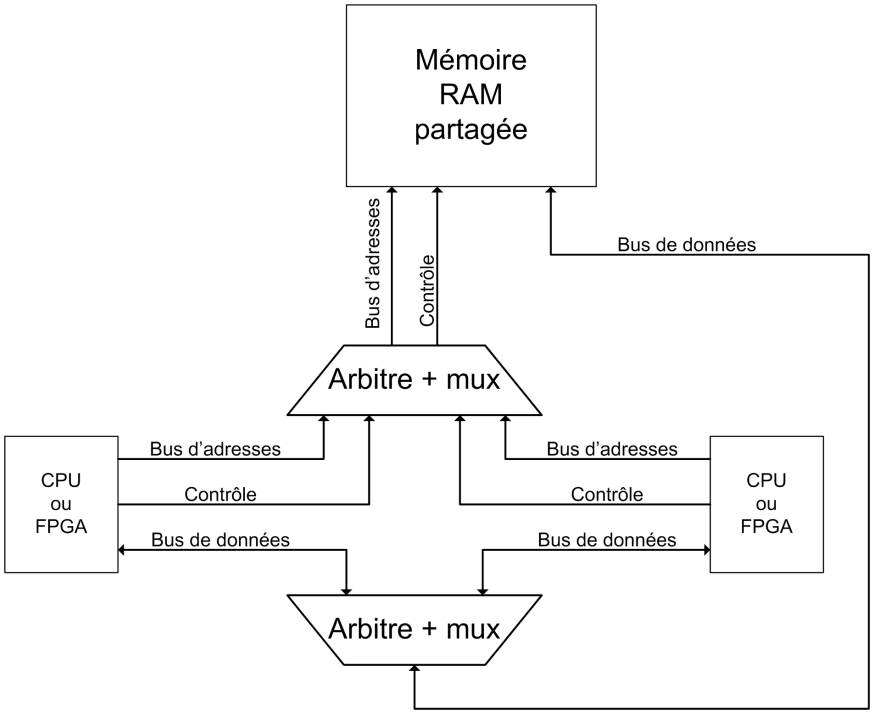

FIGURE 5.2 – Communication par mémoire simple port partagée

La communication par mémoire partagée présente l'inconvénient d'augmenter la complexité matérielle de l'architecture puisqu'il faut s'interfacer avec les bus mémoires de part et d'autre de la communication.

5.3.2 Communication par passage de message

Les différents composants d'un système multi-composants peuvent communiquer entre eux en se passant des messages. Ce type de communication suit un modèle de communication explicite. Ces messages sont alors transmis à travers une liaison point à point ou multi-point avec ou sans routeur. Le composant émetteur utilise une macro "send" pour envoyer la donnée au récepteur qui utilise une macro "receive" pour la lire. Ses deux macro doivent être spécifiés par l'utilisateur et dépendent de la nature du composant. L'opération de réception peut être bloquante ou pas. Il faut utiliser un mécanisme de synchronisation entre émetteur et récepteur pour garantir l'ordre d'exécution envoie puis réception.

La communication par passage de messages peut être parallèle (figure 5.3). Dans ce cas la liaison de données utilisée peut transférer plusieurs bits simultanément. Ainsi la durée de passage du message est diminuée mais la taille des liaisons augmente.

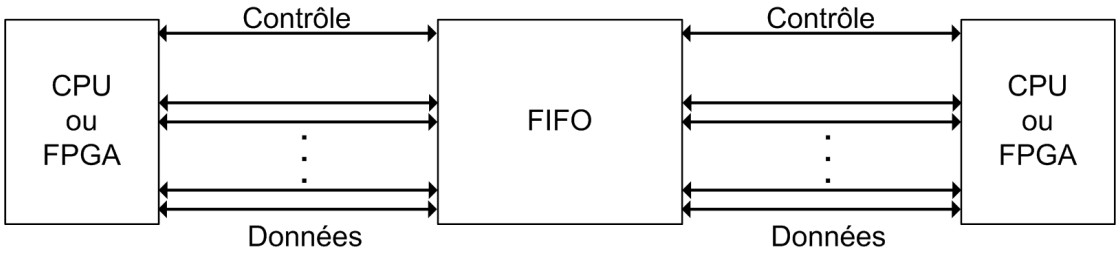

FIGURE 5.3 – Communication par passage de messages parallèle

Cette communication peut être aussi en série (figure 5.4). Dans ce cas la liaison de données utilisée ne peut transférer qu'un seul bit à la fois. Donc les différents bits du message à transmettre sont envoyés séquentiellement. Ce qui induit une augmentation de la durée de passage du message contre une diminution de la taille des liaisons.

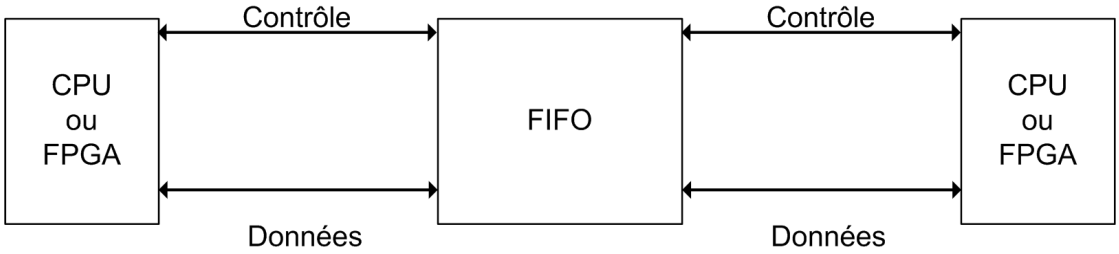

FIGURE 5.4 – Communication par passage de messages série

Pour les architectures utilisant la communication par passage de message, la FIFO est généralement projetée coté FPGA et processeur.

5.4 IP de communication proposée

Dans cette thèse, nous souhaitons étendre AAA pour couvrir les architectures mixtes. Parmi ces architectures, nous venons de voir qu'il existaient différentes techniques de communication entre les parties programmables et reconfigurables. Afin de valider nos travaux, nous ne pouvions implanter toutes ces techniques mais en choisir au moins une. Nous avons donc décidé d'implanter la communication par passage de message qui semble suffisamment générique et représentative. Nous aurions pu implanter la communication par mémoire partagée.

5.4.1 Protocole de communication

Parmi les différents types de communications par passage de messages, nous utilisons le protocole de synchronisation requête-acquittement. En effet, ce protocole permet la synchronisation des composants indépendamment de leurs fréquences de fonctionnement. Ce protocole utilise deux bits de commande pour chaque direction de transfert. Les détails d'utilisation de la liaison de communication sont indiqués sur la figure 5.5. Cette figure montre que quatre bits du bus bidirectionnel de communication sont utilisés pour les signaux de synchronisation : deux pour la synchronisation d'envoi (Req_t_cpu, Ack_f_cpu) et deux pour la synchronisation de réception (Req_f_cpu, Ack_t_cpu).

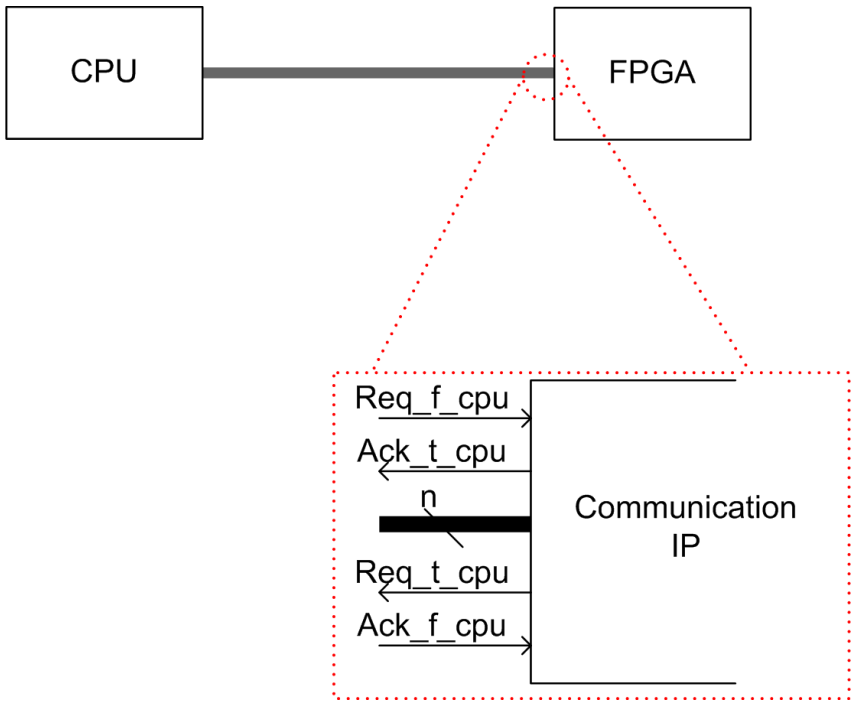

FIGURE 5.5 – Détails d'utilisation du bus de communication

Le composant émetteur déclenche le transfert. Il met les données sur le bus de communication parallèle de n bits. Puis, il envoie un signal de requête vers le récepteur. A la réception de ce signal de requête, le récepteur lit les données, les enregistres et envoie un signal d'acquittement pour indiquer la fin du transfert. Lorsque l'émetteur reçoit le signal d'acquittement, il remet à zéro la requête. Enfin, le récepteur met le signal d'acquittement à un niveau bas. Un nouveau transfert de paquet peut commencer. Le schéma temporel du signal pour une réception de paquet et un envoie paquet est indiqué dans la figure 5.6.

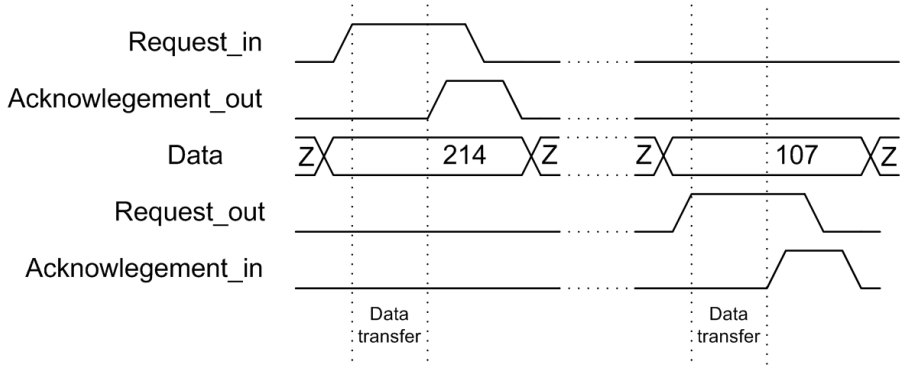

FIGURE 5.6 – Diagramme du protocole de communication

5.4.2 Architecture proposée

L'architecture de l'IP de communication que nous proposons, est composée de plusieurs modules. Les composants utilisés sont des compteurs, un comparateur, une ROM, un bloc pour gérer la synchronisation des opérateurs, une machine à états finis (FSM), des multiplexeurs et des dé-multiplexeurs. Dans ce paragraphe, nous allons présenter chacun

de ces modules puis passer à la présentation de l'architecture de l'IP de communication. Tous les ports présentés seront écrits en *italique* et suivis entre parenthèses par leurs tailles en nombre de bits.

Compteur

Le compteur que nous utilisons est représenté par la figure 5.7. Il a deux ports d'entrées : *init* (1) pour l'initialisation et *Clk* (1) pour l'incrémentation de la valeur de l'output et un port de sortie q(output_width). Ce composant dépend de trois paramètres :

* initialisation : c'est la valeur que prend tous les bits de sortie après initialisation. Ce paramètre peut prendre la valeur '1' ou '0'.
* modulo : c'est le nombre de coups d'horloge nécessaires pour que la sortie du compteur retourne à sa valeur initiale sans activer le signal de remise à zéro.
* output_width : c'est la taille de la sortie du compteur en nombre de bits.

FIGURE 5.7 – Le composant compteur

Le compteur fonctionne de la manière suivante : si le signal "init" est actif, tous les bits de la sortie prennent la valeur du paramètre initialisation. Sinon, à chaque front montant la valeur de la sortie est incrémentée jusqu'à atteindre la valeur du paramètre modulo. Si la valeur du paramètre modulo est atteinte, alors la sortie est réinitialisée.

Comparateur

Le comparateur possède, comme le montre la figure 5.8, quatre ports d'entrées : les deux données à comparer *data1* (data_width) et *data2* (data_width), le signal d'horloge *clk* (1) et le signal de remise à zéro *init* (1). Il a un seul port de sortie *out* (1).

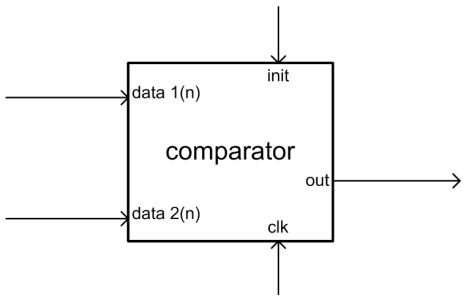

FIGURE 5.8 – Le composant comparateur

Ce composant fonctionne comme suit : si le signal "init" est à niveau haut, la sortie prend '0'. Dans le cas contraire, à la détection d'un front montant de l'horloge "clk", la sortie prend '1' si les deux valeurs des entrées "data1" et "data2" sont égales et '0' si elles ne le sont pas.

Le comparateur dépend d'un seul paramètre 'data_widht' qui fixe la taille des ports d'entrées.

ROM

Il faut disposer d'une mémoire pour stocker la séquence de communication générée dans la phase de distribution-ordonnancement. Nous l'appellerons "ROM" (Read Only Memory) dans la mesure où son contenu ne change pas pendant la durée de l'application.

Chaque case de cette mémoire contient donc un mot codant la direction du transfert et la taille des données à transférer par l'IP de communication. Le détail est donné page 104.

Cette mémoire possède un port d'entrée *adr* (adr_width) et un port de sortie *q* (mem_width) (figure 5.9). Elle dépend de trois paramètres :

* mem_size : ce paramètre fixe la taille en nombre de mots que contient la mémoire.
* mem_width : il fixe la taille en nombre de bits des mots que contient la mémoire.
* adr_width : ce paramètre fixe la largeur en nombre de bits du bus adresse de la mémoire.

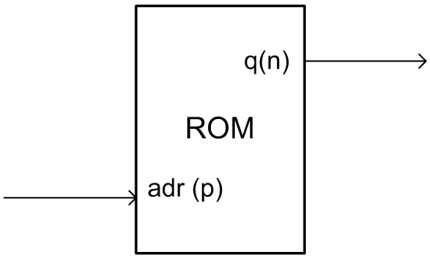

FIGURE 5.9 – Le composant ROM

A chaque instant la sortie q prend la valeur enregistrée dans la case mémoire pointée par le bus adresse.

Machine à états finis

L'ensemble de l'IP de communication est régit par une machine à états finis.

La machine à états finis est représentée par la figure 5.10. Elle a cinq ports de sortie : *start* (1), *reset* (1), *ack_t_cpu* (1), *req_t_cpu* (1) et *wr* (1). Ce composant possède aussi sept ports d'entrées : *ack_f_cpu* (1), *req_f_cpu* (1), *in_out* (1), *end* (1), *init* (1), *req_f_oprd* (1) et *clk* (1).

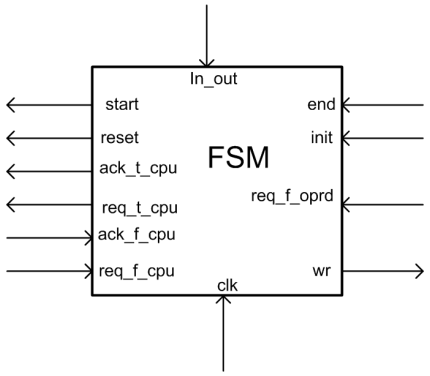

FIGURE 5.10 – Architecture de la machine à états finis

Tableau 5.1 – Valeurs des ports de sorties pour chaque état

Etat	Valeur des ports de sortie
S_{idle}	wr= 0 ,start= 0 , reset = 0 , req_t_cpu = 0 , ack_t_cpu= 0
$S_{receive1}$	wr= 1 , start= 1 , reset = 0 , req_t_cpu = 0 , ack_t_cpu= 1
$S_{receive2}$	wr= 0 , start= 0 , reset = 0 , req_t_cpu = 0 , ack_t_cpu= 0
S_{reset}	wr= 0 , start= 0 , reset = 1 , req_t_cpu = 0 , ack_t_cpu= 0
S_{send1}	wr= 0 , start= 0 , reset = 0 , req_t_cpu = 1 , ack_t_cpu= 0
S_{send2}	wr= 0 , start= 1 , reset = 0 , req_t_cpu = 0 , ack_t_cpu= 0

Les conditions de passage d'un état à un autre sont représentées dans la figure 5.11 et les valeurs que prennent les différents ports de sortie à chaque état sont regroupées dans le tableau 5.1.

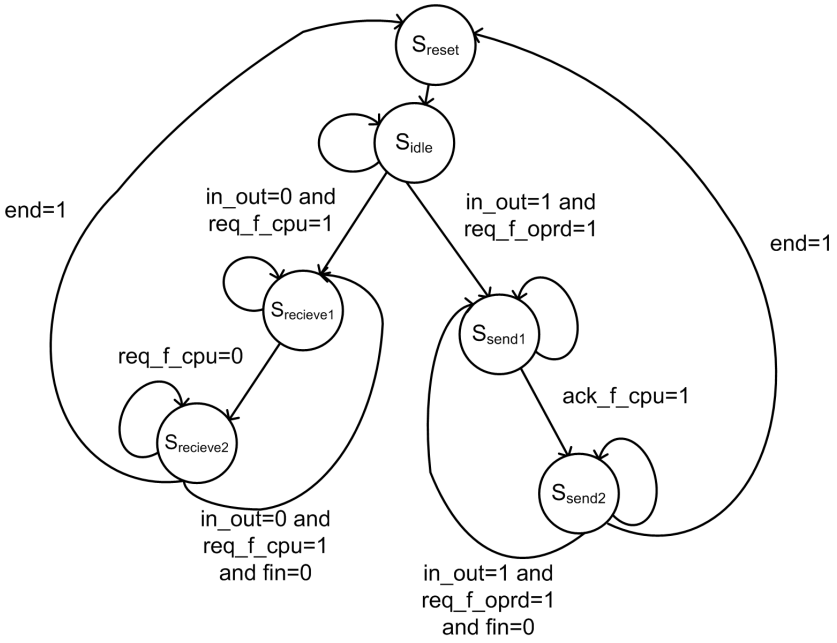

FIGURE 5.11 – Conditions de passage d'états de la machine à états finis

La machine à états finis fonctionne comme suit : à la réception du signal "init", elle bascule à l'état S_{idle}. Si le signal "init" est désactivé (à niveau bas), selon l'état courant, on teste les conditions de passage à un autre état voisin (représenté dans la figure 5.11). Par exemple si l'état courant est S_{idle}, si le signal "in_out"=0 et le signal "req_f_cpu"=1 alors on bascule à l'état $S_{receive1}$. Mais, si à partir du même état S_{idle} on reçoit le signal "in_out"=1 et "req_f_oprd"=1 la machine bascule à l'état S_{send1}. Ce basculement d'état est synchronisé sur front montant de l'horloge "clk". Après chaque changement d'état, les signaux de sortie de la machine sont changés pour correspondre aux signaux de la case correspondante au nouvel état du tableau 5.1.

mux_in_out

Il nous faut maintenant un composant permettant d'aiguiller les données vers ou depuis le FPGA. C'est le rôle du composant mux_in_out.

Le composant mux_in_out (figure 5.12) possède quatre ports : deux ports d'entrées

data_ in (bus_width) et *sel* (1), un port de sortie *data_ out* (bus_width) et un port d'entrée/sortie *com_ bus* (bus_width).

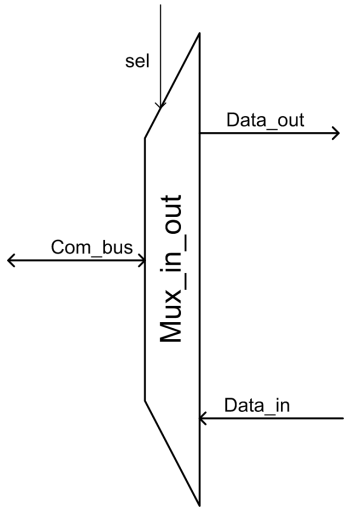

FIGURE 5.12 – Architecture du mux_in_out

Ce composant spécifie le sens de transfert des données : si le signal "sel" = 1, alors le transfert se fait du *data_ in* vers *com_ bus* et si "sel" = 0, le transfert se fait de *com_ bus* vers *data_ out*.

oprd_synch

Le rôle de ce composant est de gérer la synchronisation entre l'IP de communication et les opérateurs de calcul qui lui sont connectés. Son architecture est représentée par la figure 5.13. Il possède sept ports d'entrées : *synchro* (1), *in_ out* (1), *en* (1), *req_f_ oprd* (1), *ack_f_ oprd* (1), *init* (1) et *clk* (1). Il a quatre ports de sortie : *ack_ t_ oprd* (1), *req_ t_ oprd* (1), *req_ demux_ cmd* (req_sen_width) et *ack_ demux_ cmd* (ack_sen_width).

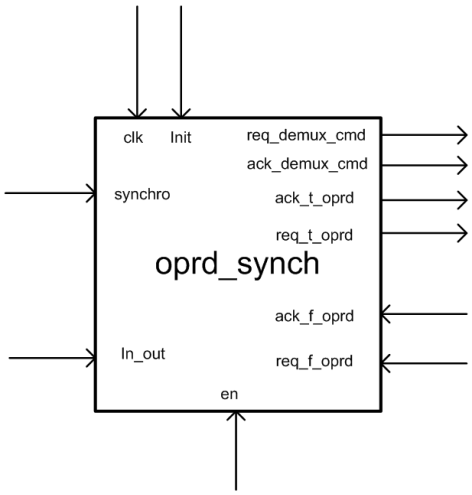

FIGURE 5.13 – Architecture du oprd_synch

Ce composant utilise deux compteurs : compt_send_req et compt_send_ack pour générer respectivement les signaux des sorties *req_ demux_ cmd* et *ack_ demux_ cmd*. Il

utilise deux signaux internes d'un bit chacun : "ack" et "req". A la réception d'un signal "init", ces signaux internes et les sorties "req_t_oprd" et "ack_t_oprd" sont remis à zéro. Les deux compteurs sont alors initialisés avec '1' pour chaque bit de leurs sorties. Si le signal "en" n'est pas actif deux cas se posent : si "synchro"=1 et "in_out"=0 alors le signal interne "req" prend '1', si "synchro"=1 et "in_out"=1 alors le signal interne "ack" prend '1'. Si le signal "en" est actif alors les sorties "ack_t_oprd" et "req_t_oprd" sont connectées respectivement aux signaux internes "ack" et "req". Si le signal "ack_f_oprd" est actif alors les signaux "req" et "req_t_oprd" sont remis à zero. Si le signal "req_f_oprd"=0 alors les signaux "ack" et "ack_t_oprd" sont remis à zero. Les compteurs compt_send_req et compt_send_ack sont respectivement incrémentés à chaque front mentant des signaux "req" et "ack".

demux

Nous utilisons un démultiplexeur générique d'un bit d'entrée (*input* (1)) et dont le port de sortie est *output* (output_width). Son port de sélection est *sel* (sel_width). Le signal d'entrée est connecté à chaque instant au bits de la sortie dont le numéro est codé par l'entrée "sel".

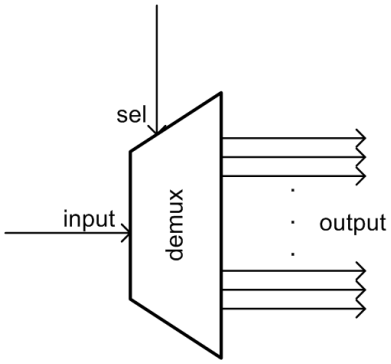

FIGURE 5.14 – Architecture du démultiplexeur

rec_reg

Ce composant est un registre de taille fixée par le paramètre reg_width. Il possède pour les données un port d'entrée *data_in* (reg_width) et un port de sortie *data_out* (reg_width) en plus de deux ports d'entrées contrôle *wr* (1) et *clk* (1). A la détection d'un front montant du signal "clk", si le signal "wr" est actif alors le signal à l'entrée "data_in" est enregistré. Le signal de sortie "data_out" prend tout le temps la valeur enregistrée dans le registre.

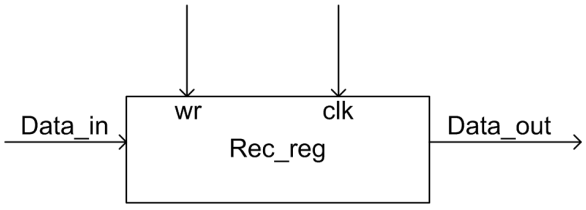

FIGURE 5.15 – Architecture du registre

mux_generic

Ce composant est un multiplexeur générique. Il a nbr_input port d'entrée données *input* (output_width) et un port de sortie données *output* (output_width). Le port d'entrée de contrôle est *sel* (sel_width).

Ce multiplexeur est constitué de output_width multiplexeurs "nbr_input vers 1".

IP de communication

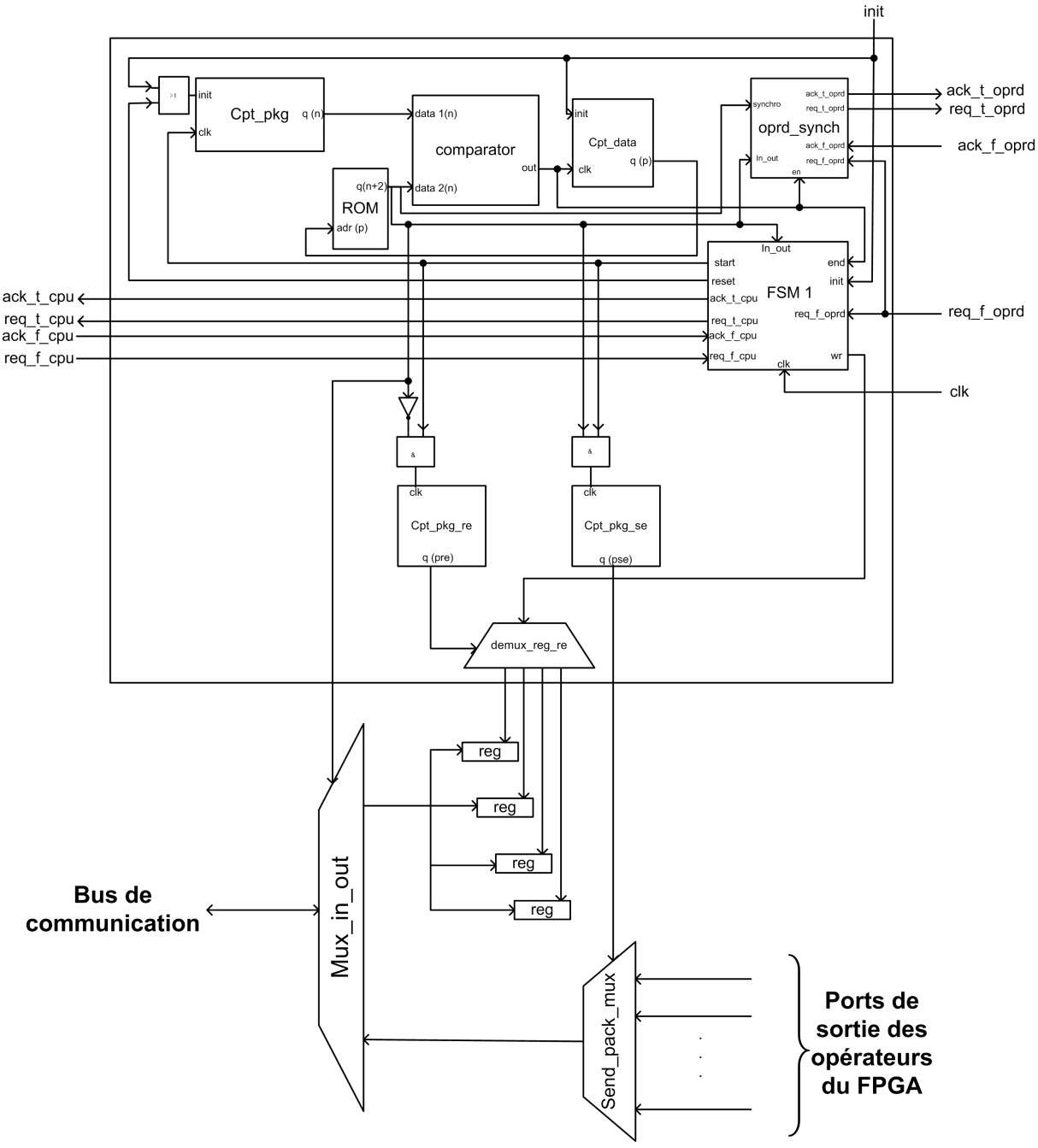

FIGURE 5.16 – Architecture de l'IP de communication

L'architecture de l'IP de communication que nous proposons est représentée dans la figure 5.16. Sur cette figure, nous n'avons pas représenté la connexion des composants

avec les entrées clk et init pour des raisons de clarté. Les composants de cette IP sont :

* cpt_pkg : c'est un compteur modulo ('max_pck_nbr'+1) dont tous les 'max_pck_nbr' bits de sortie prennent '0' à l'initialisation. Ce compteur est utilisé pour compter le nombre de paquets de la donnée courante transférée.
* ROM : c'est une mémoire contenant 'data_nbr' mots de 'max_pck_width'+2 bits chacun et dont la taille du bus adresse est 'data_nbr_width'. Cette mémoire contient dans la $i^{ème}$ case les informations correspondantes à la $i^{ème}$ données à transférer. Les 'max_pck_width' premiers bits représentent le nombre de paquets qui constituent la donnée. Le ('max_pck_width'+1)^{ème} bit est égal à '0' si l'opération est une réception de données et '1' si c'est un envoie. Le dernier bit de chaque mot enregistré dans cette mémoire indique si un signal de synchronisation doit être envoyé aux opérateurs à la suite de la fin de l'opération de transfert de la donnée correspondante.
* comparator : c'est un comparateur dont les entrées sont sur 'max_pck_width' bits. Il est utilisé pour comparer le nombre de paquets de la donnée courante transférée et le nombre total de paquets qu'elle contient pour indiquer si le transfert total de la donnée est effectué.
* cpt_data : c'est un compteur modulo ('data_nbr'+1) dont les 'data_nbr_width' bits de sorties sont mis à '0' lors de l'initialisation. Ce compteur indique le numéro de la donnée courante. Sa sortie est connectée au bus adresse de la mémoire ROM.
* synch : ce composant gère les signaux de synchronisation avec les différents opérateurs dégénérés connectés à l'IP de communication. C'est une instance du composant oprd_synch présenté dans le paragraphe 5.4.2 auquel on connecte deux démultiplexeurs (comme le montre la figure 5.17). Les ports de sortie req_t_oprd et ack_t_oprd sont de taille paramétrée respectivement par 'nbr_req_sen' et 'nbr_ack_sen'.

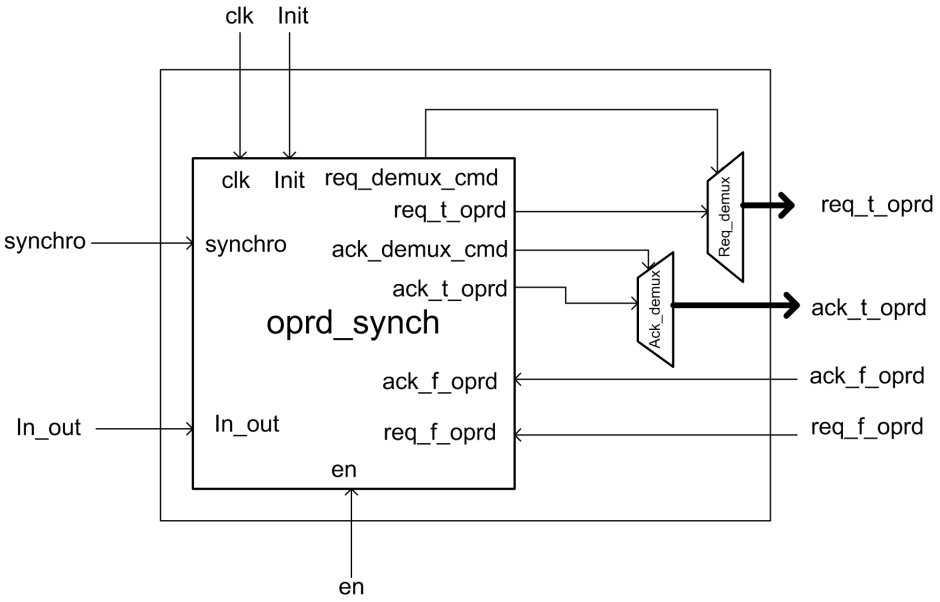

FIGURE 5.17 – Architecture du composant synch

* FSM1 : c'est une machine à états finis (paragraphe 5.4.2) qui gère le fonctionnement de l'IP de communication. Elle génère les signaux pour l'incrémentation et la remise à zéro du compteur cpt_pkg, la commande permettant la sauvegarde des données reçues et les signaux de synchronisation avec les autres composants.
* Mux_in_out : ce composant est utilisé pour choisir le sens de transfert de données (réception ou envoie). La taille des ports données est fixée par le paramètre 'com_bus_width'.
* Send_pack_mux : c'est un multiplexeur à 'sen_pck_nbr' entrées de 'com_bus_width' bits chacun et de taille de port sel égal à 'sen_pck_nbr_width'. Il sert à sérialiser l'envoie de données en choisissant le paquet à envoyer.
* Reg : ce sont des registres de taille 'com_bus_width' chacun. Ils sont utilisés pour sauvegarder les paquets reçus.
* cpt_pkg_se : c'est un compteur modulo 'sen_pck_nbr' dont les 'sen_pck_nbr_width' bits de sortie sont initialisés à '1'. Il est utilisé pour compter le nombre de paquets envoyés.
* cpt_pkg_re : c'est un compteur modulo 'rec_pck_nbr' dont les 'rec_pck_nbr_width' bits de sortie sont initialisés à '1'. Il est utilisé pour compter le nombre de paquets reçus.
* demux_reg_re : c'est un démultiplexeur à 'rec_pck_nbr' sorties de un bit chacune. Son entrée sel est de taille 'rec_pck_nbr'. Il sert à aiguiller le signal de commande au registre correspondant.

L'IP de communication que nous proposons dépend de plusieurs paramètres :

* max_pck_nbr : le nombre de paquets que contient la plus grande donnée à transférer. Il spécifie la taille des mots enregistrés dans la ROM et le modulo du compteur cpt_pck.
* max_pck_width : le nombre de bits nécessaires pour coder le paramètre 'max_pck_nbr'. Il spécifie la taille de la sortie de la ROM, de la sortie du cpt_pck et des entrées du comparateur.
* data_nbr : le nombre de données à transférer dans les deux sens. Il spécifie la taille de la ROM et le modulo du cpt_data.
* data_nbr_width : le nombre de bits nécessaires pour coder le paramètre 'data_nbr'. Il fixe la taille de la sortie du compteur compt_data et la taille du bus adresse de la ROM.
* rec_pck_nbr : le nombre total de paquets à recevoir. Il spécifie le modulo du compteur cpt_pck_re, le nombre de sortie du démultiplexeur demux_reg_re et le nombre de registres utilisés.
* rec_pck_nbr_width : le nombre de bits nécessaires pour coder le paramètre 'rec_pck_nbr'. Il fixe la taille de la sortie du compteur cpt_pck_re et la taille du port sel du démultiplexeur demux_reg_re.
* sen_pck_nbr : le nombre total de paquets à envoyer. Il spécifie le modulo du compteur cpt_pck_se et le nombre de sortie du multiplexeur send_pack_mux.
* sen_pck_nbr_width : le nombre de bits nécessaires pour coder le paramètre 'sen_pck_nbr'. Il fixe la taille de la sortie du compteur cpt_pck_se et la taille du port sel du multiplexeur send_pack_mux.

* com_bus_width : le nombre de bits du bus de communication réservés pour le transfert de données. Il spécifie la taille du paquet, la taille des ports (autre que le port sel) du mux_in_out, la taille des registres et la taille des ports du multiplexeur send_pack_mux autre que le port sel.
* data_in_width : le nombre de bits de tous les paquets à envoyer. Il spécifie la taille du port dat_in de l'IP de communication.
* data_out_width : le nombre de bits de tous les paquets à recevoir. Il spécifie la taille du port data_out de l'IP de communication.
* nbr_req_sen : le nombre d'opérateurs implémentés dans le FPGA pour lequel il faut envoyer un signal requête. Ce paramètre spécifie la taille du port req_t_oprd de l'IP de communication, le modulo du compteur compt_send_req du oprd_synch et le nombre de ports de sortie du démultiplexeur req_demux.
* req_sen_width : le nombre de bits nécessaires pour coder le paramètre nbr_req_sen. Il spécifie la taille de la sortie du compteur compt_send_req et du port req_demux-_cmd du composant oprd_synch et de l'entrée sel du démultiplexeur req_demux.
* nbr_ack_sen : le nombre d'opérateurs implémentés dans le FPGA pour lequel il faut envoyer un signal acquittement. Ce paramètre spécifie la taille du port ack_t_oprd de l'IP de communication, le modulo du compteur compt_send_ack du oprd_synch et le nombre de ports de sortie du démultiplexeur ack_demux.
* ack_sen_width : c'est le nombre de bits nécessaires pour coder le paramètre nbr-_ack_sen. Il spécifie la taille de la sortie du compteur compt_send_ack et du port ack_demux_cmd du composant oprd_synch et de l'entrée sel du démultiplexeur ack_demux.

Cette IP est générée pour exécuter une liste de communications prédéfinie. La ROM contient les informations concernant cette liste d'opérations de communication à effectuer. Si l'opération courante est une réception de données, alors le signal in_out à l'entrée du FSM est à '0', à la réception d'une requête du composant distant (req_f_cpu=1) le FSM met les signaux wr, start et ack_t_cpu à '1' pour permettre respectivement l'enregistrement du paquet reçu dans un registre, incrémenter les compteurs cpt_pkg_re et cpt_pkg et l'envoie d'acquittement au composant distant. Le retour du signal req_f_cpu à '0' indique la fin de réception du paquet courant, tous les signaux de sortie de la FSM sont remis à '0'. Si l'intégralité de la donnée courante est reçue, alors on a égalité des valeurs à l'entrée du comparateur (signaux de sortie du cpt_pkg et ROM), donc la sortie du comparateur sera à '1'. Par conséquent, le compteur cpt_data sera incrémenté et on passe à l'opération de communication suivante (on passe à la case suivante de la ROM) et la FSM reçoit '1' en son port d'entrée end et envoie le signal reset pour réinitialiser le compteur cpt_pkg. S'il reste encore des paquets à recevoir, alors à la réception d'une nouvelle requête req_f_cpu un nouveau cycle commence. Si l'opération à effectuer est un envoi, alors le signal in_out est égal à '1' ; à la réception d'une requête de l'opérateur contenu dans le FPGA (req_f_oprd=1), la FSM envoie une requête au composant distant (req_t_cpu=1). Puis à la réception d'un signal d'acquittement du composant distant (ack_f_cpu=1), la FSM met le signal start à '1' pour incrémenter les compteurs cpt_pkg et cpt_pkg_se.

Nous avons synthétisé cette architecture de l'IP de communication pour obtenir un ordre de grandeur de la taille occupée sur un FPGA et pour valider son fonctionnement. Nous avons choisi une taille de bus de communication de 8 bits. Dans ce premier exemple

Tableau 5.2 – Détails des données transférées pour le premier exemple

Numéro	Sens	Nombre de paquets	Synchronisation	Case mémoire
1	réception	3	0	0000011
2	réception	3	1	1000011
3	envoie	5	1	1100101
4	réception	18	1	1010010
5	envoie	6	1	1100110
6	réception	2	0	0000010
7	réception	1	1	1000001
8	envoie	3	1	1100011

simple, l'IP de communication est conçue pour le transfert de 8 données dont les détails sont regroupés dans le tableau 5.2. Ce tableau présente pour chaque donnée le numéro, le sens du transfert, le nombre de paquets qu'elle contient, la nécessité d'envoi du signal de synchronisation à l'OPRd et le contenue de la case mémoire correspondante. Le nombre total de paquets à transférer est 41 : 27 paquets à recevoir et 14 à envoyer. La plus grande donnée à transmettre est constituée de 18 paquets. Pour la synchronisation, 3 requêtes et 3 acquittements doivent être envoyés.

Le FPGA cible est Stratix EP1S40F780C5. Ce FPGA comporte 41250 éléments logiques, ce qui représente un nombre d'éléments logiques plus faible que la plus part des FPGA mis sur le marché par les deux plus grands fabricants (Altera et Xilinx) [Xilinx, b] [Altera, b] mais suffisants pour notre exemple. Cette IP de communication utilise 306 éléments logiques, donc moins de 1 % de la surface du FPGA. Ainsi, l'utilisation de cet IP de communication laisse un grand espace pour les composants de calcul.

5.4.3 Transformation de macro-exécutables en séquence de communications

Nous nous intéressons aux séquences de communications contenues dans les macros générés pour chaque opérateur de communication que comporte un FPGA après transformation du graphe d'architecture (paragraphe 4.8.1). Chacune de ces macros contient deux listes de communications : une concerne les communications de cet opérateur de communication avec l'opérateur distant (processeur ou autre FPGA) et le second concerne la communication entre l'opérateur de communication et les opérateurs de calcul contenus dans le même FPGA. Ces deux listes de communications sont fortement liées puisque l'opérateur de communication ne fait pas de calcul, il gère tout simplement le transfert de données entre les opérateurs qui lui sont connectées. Donc, pour chaque réception contenue dans une liste, correspond un envoie dans l'autre liste. Nous allons nous intéresser aux macros d'envoi et de réception de données. Ces macros sont de la forme *send_ (_ nom port source, def operateur émetteur, nom operateur émetteur, nom operateur récepteur)* pour l'envoie et *recv_ (_ nom port source, def operateur émetteur, nom operateur émetteur, nom operateur récepteur)* pour la réception de données. Pour pouvoir générer cette IP de communication et la connecter correctement aux différents circuits de calcul et de contrôle contenus dans le FPGA. Nous devons établir la liste des communications à effectuer et avoir des informations à propos des ports émetteurs, des ports récepteurs, de la taille de chaque donnée à transmettre et la nécessité d'envoyer un signal de synchronisation (requête ou acquittement). Nous parcourons alors la première série de macros de

communications (celle qui concerne la communication avec l'opérateur distant) en cherchant les macros *recv_* ou *send_*. Le macro utilisé indique le sens de transfert. Si c'est un envoie, le nom du port source et du composant destinataire sont contenus dans le macro. Par contre dans le cas d'une réception, il faut chercher le nom du port destinataire dans la macro d'envoi qui lui correspond (dans la deuxième liste de macros). Les tailles des données sont obtenues à partir des macros d'allocation de mémoire. Ainsi, on obtient une liste des communications à effectuer en respectant l'ordre des macros. En parcourant cette liste, on peut identifier les dernières communications établies vers (respectivement depuis) un opérateur de calcul pour indiquer qu'il faut lui envoyer un signal de requête (respectivement acquittement). Une fois cette liste établie, on demande à l'utilisateur de spécifier le nombre de bits du bus de communication réservé aux données et on calcule tous les paramètres de l'IP de communication.

La figure 5.18 représente un exemple de fichier macros généré pour une IP de communication.

A partir des macros d'allocation de mémoire, on conclut qu'il y a cinq données à transférer et on déduit la taille de chacune d'elles. Puis en parcourant le premier thread de communication, on établit la liste des communications. Dans cet exemple, la première communication est une réception de la donnée "*_algoMain_TOP_out*". A partir du second thread, on établit que cette première donnée reçue sera envoyée à l'opérateur dégénéré "DC_val_circuit1". Le nom du port de réception de cette donnée est connu à partir des liaisons spécifiées dans le graphe d'algorithme. De la même façon, on établit que la seconde communication est la réception de la donnée "*_algoMain_LEFT_out*" et que cette donnée sera envoyée aux opérateurs dégénérés "DC_val_circuit1" et "SAD_H_circuit3". La troisième est un envoi de "*_algoMain_calc_dc_val_DC_VAL*" à partir de l'opérateur dégénéré "DC_val_circuit1". La quatrième est une réception de "*_algoMain_SRC_SRC*" qui sera envoyée vers l'opérateur dégénéré "SAD_H_circuit3". La dernière communication est un envoi de "*_algoMain_SAD_h_SAD_h*" à partir de l'opérateur dégénéré "SAD_H_circuit3". Ainsi la liste des communications de cet exemple est établie.

On peut passer maintenant à la génération du VHDL. Un dossier nommé VHDL est créé dans le dossier contenant le fichier sdx spécifié par l'utilisateur. On y copie alors les fichiers VHDL des composants qui ne nécessitent pas de modification. Puis, on intègre les valeurs des différents paramètres dans le fichier top level "com.vhd". Enfin, le fichier "rom.vhd" contenant la description de la mémoire est créé en utilisant les informations du dictionnaire communication (pour plus d'informations voir l'annexe A).

5.5 Générations d'exécutifs pour les architectures mixtes

Le générateur d'exécutifs de SynDEx est utilisé pour générer les exécutifs correspondants aux composants programmables ainsi que les listes de macros de communications correspondantes à chaque IP de communication. Le générateur de code de SynDEx-IC est utilisé pour générer un fichier VHDL pour chaque sous-graphe d'opérations distribué sur FPGA. Puis, les différents sous-graphes distribués sur le même FPGA sont regroupés et connectés à l'IP de communication. Cela s'effectue en supprimant les différents actuators et sensors et en reconnectant leurs successeurs et prédécesseurs à l'IP de communication.

```
alloc_(uchar,_algoMain_SRC_SRC,256)
alloc_(uchar,_algoMain_TOP_out,16)
alloc_(uchar,_algoMain_LEFT_out,16)
alloc_(int,_algoMain_SAD_h_SAD_h,1)
alloc_(uchar,_algoMain_calc_dc_val_DC_VAL,1)
thread_(TCP,x,com_fpga_TCP,cpu)
  loadFrom_(cpu,x_fpga)
  Pre1_(_algoMain_calc_dc_val_DC_VAL_com_fpga_TCP_x_empty,x_fpga,_algoMain_calc_dc_val_DC_VAL,empty)
  Pre1_(_algoMain_SAD_h_SAD_h_com_fpga_TCP_x_empty,x_fpga,_algoMain_SAD_h_SAD_h,empty)
  loop_
    Suc1_(_algoMain_TOP_out_com_fpga_TCP_x_fpga_empty,x_fpga,_algoMain_TOP_out,empty)
    recv_(_algoMain_TOP_out,CPU,cpu,com_fpga_TCP)
    Pre1_(_algoMain_TOP_out_com_fpga_TCP_x_fpga_full,x_fpga,_algoMain_TOP_out,full)
    Suc1_(_algoMain_LEFT_out_com_fpga_TCP_x_fpga_empty,x_fpga,_algoMain_LEFT_out,empty)
    recv_(_algoMain_LEFT_out,CPU,cpu,com_fpga_TCP)
    Pre1_(_algoMain_LEFT_out_com_fpga_TCP_x_fpga_full,x_fpga,_algoMain_LEFT_out,full)
    Suc1_(_algoMain_calc_dc_val_DC_VAL_com_fpga_TCP_x_full,x_fpga,_algoMain_calc_dc_val_DC_VAL,full)
    send_(_algoMain_calc_dc_val_DC_VAL,com_fpga_TCP,com_fpga_TCP,cpu)
    Pre1_(_algoMain_calc_dc_val_DC_VAL_com_fpga_TCP_x_empty,x_fpga,_algoMain_calc_dc_val_DC_VAL,empty)
    Suc1_(_algoMain_SRC_SRC_com_fpga_TCP_x_fpga_empty,x_fpga,_algoMain_SRC_SRC,empty)
    recv_(_algoMain_SRC_SRC,CPU,cpu,com_fpga_TCP)
    Pre1_(_algoMain_SRC_SRC_com_fpga_TCP_x_fpga_full,x_fpga,_algoMain_SRC_SRC,full)
    Suc1_(_algoMain_SAD_h_SAD_h_com_fpga_TCP_x_full,x_fpga,_algoMain_SAD_h_SAD_h,full)
    send_(_algoMain_SAD_h_SAD_h,com_fpga_TCP,com_fpga_TCP,cpu)
    Pre1_(_algoMain_SAD_h_SAD_h_com_fpga_TCP_x_empty,x_fpga,_algoMain_SAD_h_SAD_h,empty)
  endloop_
  saveUpto_(cpu,x_fpga)
endthread_
thread_(inter_oprd,x_fpga,DC_val_circuit1,SAD_H_circuit3,com_fpga_TCP)
  loadDnto_(x,DC_val_circuit1,SAD_H_circuit3)
  Pre1_(_algoMain_TOP_out_com_fpga_TCP_x_fpga_empty,x,_algoMain_TOP_out,empty)
  Pre1_(_algoMain_LEFT_out_com_fpga_TCP_x_fpga_empty,x,_algoMain_LEFT_out,empty)
  Pre1_(_algoMain_SRC_SRC_com_fpga_TCP_x_fpga_empty,x,_algoMain_SRC_SRC,empty)
  loop_
    Suc1_(_algoMain_TOP_out_com_fpga_TCP_x_fpga_full,x,_algoMain_TOP_out,full)
    send_(_algoMain_TOP_out,com_fpga_TCP,com_fpga_TCP,DC_val_circuit1)
    Pre1_(_algoMain_TOP_out_com_fpga_TCP_x_fpga_empty,x,_algoMain_TOP_out,empty)
    Suc1_(_algoMain_LEFT_out_com_fpga_TCP_x_fpga_full,x,_algoMain_LEFT_out,full)
    send_(_algoMain_LEFT_out,com_fpga_TCP,com_fpga_TCP,DC_val_circuit1,SAD_H_circuit3)
    Pre1_(_algoMain_LEFT_out_com_fpga_TCP_x_fpga_empty,x,_algoMain_LEFT_out,empty)
    Suc1_(_algoMain_calc_dc_val_DC_VAL_com_fpga_TCP_x_empty,x,_algoMain_calc_dc_val_DC_VAL,empty)
    recv_(_algoMain_calc_dc_val_DC_VAL,DC_val_fpga,DC_val_circuit1,com_fpga_TCP)
    Pre1_(_algoMain_calc_dc_val_DC_VAL_com_fpga_TCP_x_full,x,_algoMain_calc_dc_val_DC_VAL,full)
    Suc1_(_algoMain_SRC_SRC_com_fpga_TCP_x_fpga_full,x,_algoMain_SRC_SRC,full)
    send_(_algoMain_SRC_SRC,com_fpga_TCP,com_fpga_TCP,SAD_H_circuit3)
    Pre1_(_algoMain_SRC_SRC_com_fpga_TCP_x_fpga_empty,x,_algoMain_SRC_SRC,empty)
    Suc1_(_algoMain_SAD_h_SAD_h_com_fpga_TCP_x_empty,x,_algoMain_SAD_h_SAD_h,empty)
    recv_(_algoMain_SAD_h_SAD_h,SAD_H_fpga,SAD_H_circuit3,com_fpga_TCP)
    Pre1_(_algoMain_SAD_h_SAD_h_com_fpga_TCP_x_full,x,_algoMain_SAD_h_SAD_h,full)
  endloop_
  saveFrom_(x,DC_val_circuit1,SAD_H_circuit3)
endthread_
```

Macros d'allocation de mémoire

FIGURE 5.18 – Exemple de liste de macros de communication

5.6 Conclusion

Dans ce chapitre nous avons présenté l'IP générique de communication que nous proposons pour gérer les communications et synchronisation du FPGA avec les autres composants. En effet, la génération de ce composant entre dans le cadre de l'étape de génération automatique des codes mixtes et essentiellement les communications entre les différents composants de l'architecture. Cette IP est obtenue à partir des listes de macros de communication générées par SynDEx. En conséquences, les exécutifs de l'implantation de l'application sur l'architecture mixte peuvent être générés automatiquement en trois parties : les parties distribuées sur les composants programmables sont générés par SynDEx, les parties distribuées sur les composants reconfigurables sont générées par SynDEx-IC et les communications entres ces deux parties sont générés par notre nouvel outil pour

les architectures mixtes. De cette façon, on obtient un outil capable d'optimiser (chapitre 4) et de générer automatiquement les exécutifs correspondants aux applications sur les architectures mixtes.

Les travaux décrits dans ce chapitre sont publiés dans [Feki et al., 2013] et [Feki et al., 2014a].

Chapitre 6

Implantation et application

6.1 Introduction

Nous avons détaillé, dans les chapitres 4 et 5, les fondements théoriques et les aspects pratiques du couplage des outils SynDEx et SynDEx-IC pour l'optimisation du partitionnement et de l'implémentation des algorithmes sur les architectures mixtes comportant des composants programmables et d'autres reconfigurables et la génération automatique des codes correspondants. Dans ce chapitre, nous allons présenter l'implantation du couplage des outils SynDEx et SynDEx-IC (l'algorithme 3 de la page 80). La technique utilisée repose sur la connexion des outils par l'intermédiaire de fichiers et de programmes spécifiques qui manipulent les différents fichiers utilisés par SynDEx et SynDEx-IC pour les adapter à notre algorithme.

Dans la seconde partie de ce chapitre, nous présentons l'utilisation de notre outil pour optimiser l'implémentation de l'estimateur de mouvement de la norme de codage vidéo H.264 sur une architecture mixte.

6.2 Implantation du couplage

Pour faire associer SynDEx et SynDEx-IC dans le but de supporter les architectures mixtes, nous nous reposons sur des programmes écrits en langage Python. Ce langage de programmation a été choisi car c'est un langage qui utilise des structures de données évoluées et qui présente plusieurs outils de manipulations de chaines de caractères et de fichiers. Ce langage est aussi particulièrement adapté pour manipuler d'autre programmes en ligne de commande comme nous le ferons avec SynDEx et SynDEx-IC. En effet chacun d'eux a été conçu pour fonctionner sans interface graphique. Ainsi on peut les exécuter depuis un terminal (shell linux ou windows), leur passer les fichiers d'entrées, des paramètres et récupérer les résultats dans un fichier de sortie.

La figure 6.1 représente les différentes fonctions utilisées et les interactions entre elles. La fonction "generation_ip" n'y est pas représentée par soucis de clarté.

Ces programmes utilisent les fichiers de spécification créés par l'utilisateur et font appel à SynDEx et SynDEx-IC. Nous allons dans la suite de cette section détailler chacune de ces fonctions.

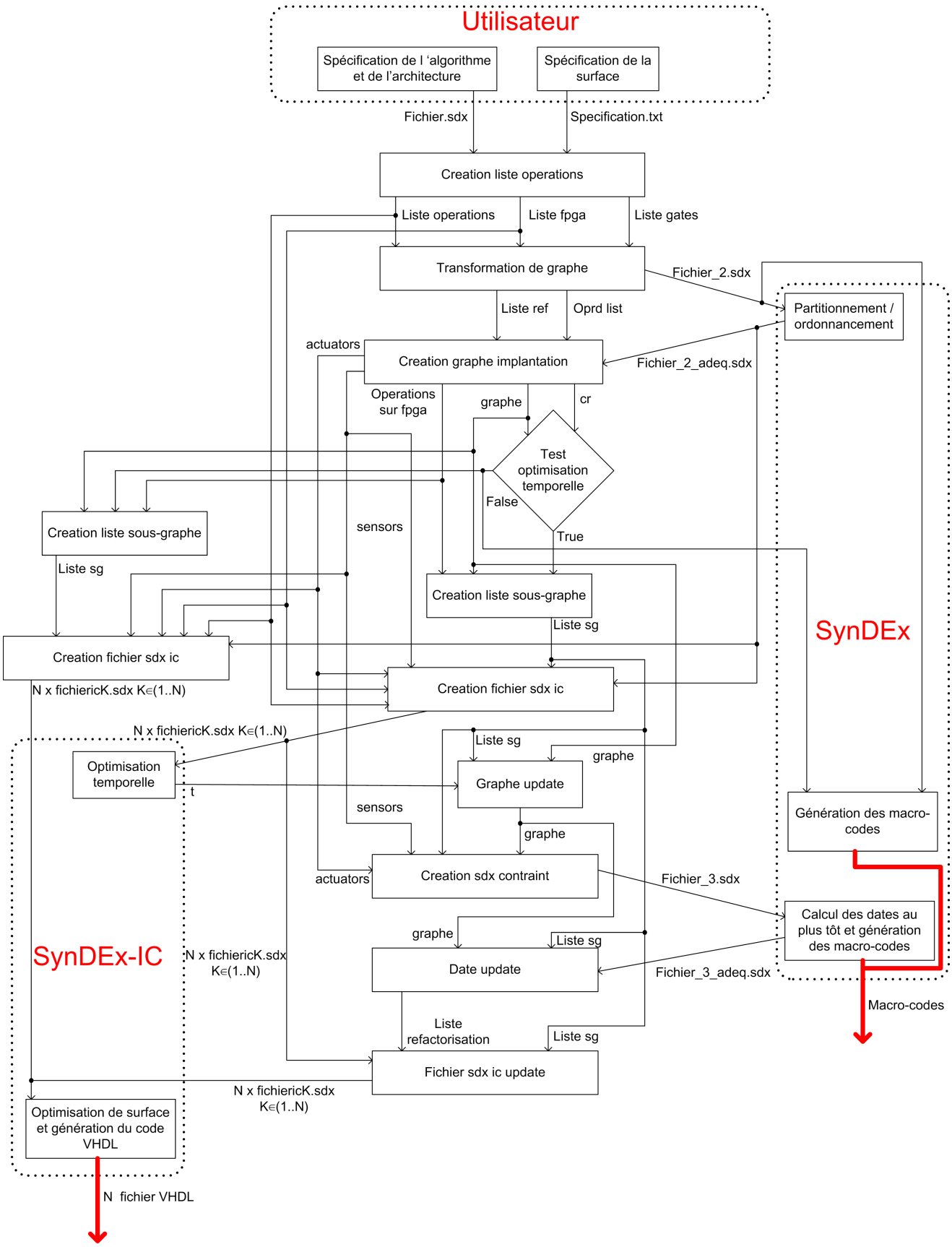

FIGURE 6.1 – Flot de couplage des outils SynDEx et SynDEx-IC

6.2.1 Creation_liste_operation

La fonction "creation_liste_operation" a pour but d'extraire et regrouper les informations nécessaires pour la transformation automatique de graphe d'architecture. Ses entrées sont les fichiers, "fichier.sdx" contenant la spécification de l'algorithme et l'architecture et "specification.txt" contenant la spécification spatiale des opérations distribuables sur le FPGA. Cette fonction produit trois listes : "liste gate", "liste operations" et "liste fpga".

La liste "liste gate" contient les définitions des ports de chaque opérateur reconfigurable de l'architecture. Pour ce faire, cette fonction parcourt les définitions des opérateurs contenues dans le fichier "fichier.sdx". A chaque définition d'un opérateur reconfigurable (opérateur dont le nom commence par fpga), elle ajoute le nom de la définition à la liste "liste fpga". Puis elle ajoute à la liste "liste gate" une liste des lignes spécifiant les ports de l'opérateur. Ces définitions de ports sont de la forme "*gate type_port nom_port*". De cette façon, on garde le même ordre des éléments entre ces deux listes. En effet, chaque élément de la liste "liste gate" correspond au nom de définition du FPGA ayant le même indice dans la liste "liste fpga".

Puis à partir de la partie spécifiant la latence des opérations sur FPGA (partie commençant par "*extra_ durations_ operator fpga*"), et pour chaque composant reconfigurable, on crée une liste "liste sdx nom fpga" de ces opérations de la forme *(nom_ opération, durée)* où *nom_ opération* est le nom de définition de l'opération et *durée* est la durée d'exécution correspondante.

Ensuite le fichier "specification.txt" est parcouru et une liste "liste txt nom fpga" contenant des éléments de la forme *(nom_ opération, surface_ occupée)* est créée pour chaque composant reconfigurable. *nom_ opération* est le nom de définition de l'opération que peut exécuter le FPGA et *surface_ occupée* est le nombre de blocs logiques que nécessite son implémentation sur le FPGA.

Puis, on crée la liste finale des opérations que peut exécuter chaque FPGA. Cette liste nommée "liste operations nom fpga" contient des éléments de la forme *(nom_ opération, durée, surface_ occupée)*. On ajoute à cette liste les données correspondantes aux opérations contenues dans les deux listes précédentes (liste sdx nom fpga et liste txt nom fpga). Puis pour les opérations de la "liste sdx nom fpga" qui n'apparaissent pas dans la "liste txt nom fpga", on demande à l'utilisateur de spécifier la surface occupée par l'opération sur le FPGA avant d'ajouter les informations à la liste "liste operations nom fpga". De la même façon, pour toute opération de la liste "liste txt nom fpga" qui n'est pas dans la liste "liste sdx nom fpga", on demande à l'utilisateur de spécifier la durée d'exécution correspondante, avant d'ajouter l'élément correspondant à la liste "liste operations nom fpga". Ensuite, toutes ces listes sont regroupées dans une liste "liste operations" en gardant le même ordre que celui de la liste "liste fpga". Ainsi, comme pour la liste "liste gate", chaque élément de la liste "liste operation" correspond à la définition de FPGA ayant le même indice dans la liste "liste fpga".

6.2.2 Transformation_de_graphe

La fonction "transformation_de_graphe" crée un nouveau fichier "fichier_2.sdx" contenant le graphe d'algorithme spécifié par l'utilisateur et le graphe d'architecture transformé. Elle reçoit en entrée le fichier "fichier.sdx" contenant l'algorithme et l'architecture spécifiées par l'utilisateur et les listes "liste gate", "liste operations" et "liste fpga" créées par la fonction "création liste operation". Cette fonction retourne une liste "liste ref" des références aux opérations que contient l'algorithme principal, une liste "oprd list" des

opérateurs de traitement et une liste "liste ip com" des opérateurs de communication créés dans les FPGAs.

Cette fonction commence par parcourir le fichier "fichier.sdx" spécifié par l'utilisateur et fait une copie des définitions d'algorithmes dans le fichier "fichier_2.sdx". Lors de la copie des références aux opérations que contient l'algorithme principal, elle crée une liste "liste ref" contenant des éléments de la forme *(nom_definition, nom_reference)*. Cette fonction fait aussi une copie des définitions que contient la partie architecture du fichier "fichier.sdx" dans "fichier_2.sdx" tout en créant une liste "fpga ref" des références à des FPGA que contient l'architecture principale, une liste "opr archi" des opérateurs, une liste "med archi" des moyens de communication et une liste "connection" des connections entre les ports des opérateurs et les moyens de communication de l'architecture principale. Puis elle crée les définitions des opérateurs qui remplacerons chaque FPGA dans l'architecture principale.

Ainsi pour chaque référence à un FPGA de la liste "fpga ref", cette fonction crée pour chaque port contenu dans la liste "liste gate" un opérateur pour la communication, pour chaque opération de la liste "liste operations" un opérateur capable de l'exécuter et un moyen de communication "inter_oprd" de type SAM multipoint supportant la diffusion matérielle. Elle crée aussi l'architecture transformée selon l'algorithme de transformation de graphe (Algorithme 4), en utilisant les informations des listes créées précédemment, tout en ajoutant dans la partie "*extra durations*" les durées d'exécution correspondantes aux opérations que peut exécuter chaque nouveau opérateur et les durées de traversée correspondantes au moyen de communication "inter_oprd" (qui sont toutes nulles).

6.2.3 Distribution/Ordonnencement

Pour la distribution et l'ordonnancement, on exécute l'heuristique de SynDEx en ligne de commande en lui fournissant le fichier "fichier_2.sdx" contenant le graphe d'algorithme spécifié par l'utilisateur et le graphe d'architecture modifié par 6.2.2 comme entrée. Le résultat de l'adéquation est sauvegardé dans le fichier "fichier_2_adeq.sdx".

On utilise la bibliothèque standard de Python "os" permettant l'utilisation de fonctionnalités dépendantes du système d'exploitation et essentiellement la fonction "os.system(str)" qui exécute le paramètre str en ligne de commande.

6.2.4 Creation_graphe_implantation

La fonction "creation_graphe_implantation" permet de créer le graphe d'implémentation contenant la distribution donnée par SynDEx et d'extraire les données nécessaires à la création des fichiers sdx destinés à SynDEx-IC.

La fonction "creation_graphe_implantation" reçoit le fichier "fichier_2_adeq.sdx" contenant l'adéquation, la liste "oprd list" des opérateurs de traitement contenus dans les FPGA et la liste "liste ref" des références d'opérations contenues dans l'algorithme principal. Elle retourne le graphe d'implantation créé "graphe", une liste "operations sur fpga" des opérations distribuées sur les composants reconfigurables, une liste "actuators" des données transférées depuis les communicateurs des FPGA, une liste "sensors" des données transférées vers les communicateurs des FPGA et la longueur du chemin critique "cr".

Cette fonction utilise une classe "node", que nous avons créée, pour modéliser les nœuds du graphe. Chaque instance de la classe "node" est caractérisée par : sa durée,

sa date de début au plus tôt, sa date de fin au plus tôt, l'opérateur qui l'exécute, ses prédécesseurs, ses successeurs de l'algorithme, ses successeurs sur l'opérateur qui l'exécute, sa date de fin au plus tard, sa date de début au plus tard. La classe node possède une méthode permettant de calculer les dates de début et de fin au plus tard du nœud qui l'appelle.

Le graphe d'implémentation est modélisé par un dictionnaire [1] "graphe" dont les clefs sont les noms des nœuds du graphe et les éléments sont des instances de la classe node. Ces nœuds (de traitement et de communication) qui constituent le graphe d'implantation, sont créés à partir de la partie "schedules" du fichier "fichier_2_adeq.sdx" en spécifiant pour chaque nœud l'opérateur sur lequel il est distribué, son successeur sur le même opérateur et ses dates de début et de fin au plus tôt. La durée d'exécution de chaque nœud est calculée à partir de ses dates au plus tôt. En même temps, cette fonction crée une liste "operations sur fpga" des références d'opérations distribuées sur FPGA, et deux listes "actuators" et "sensors" des données transférées respectivement depuis et vers le communicateur que contient le FPGA. Ces deux dernières listes regroupent les informations nécessaires pour la création des actuators et des sensors dans les fichiers sdx destinés à SynDEx-IC. Les éléments de ces listes sont de la forme (type[taille], nom_reference.nom_port). Puis les successeurs et les prédécesseurs de chaque nœud sont ajoutés à partir de la partie "dependences" de la spécification du graphe d'algorithme. Cette fonction calcule aussi la longueur du chemin critique "cr" qui est la plus grande valeur de date de fin au plut tôt.

6.2.5 Test_optimisation_temporelle

La fonction "test_optimisation_temporelle" a pour but de tester l'existence d'opérations distribuées sur FPGA appartenant au chemin critique. Elle reçoit le graphe d'implémentation, la liste "oprd list" des opérateurs dégénérés et la valeur de la longueur du chemin critique "cr". Elle retourne une variable booléenne "test".

La fonction "test_optimisation_temporelle" commence par calculer les dates aux plus tard de tous les nœuds du graphe d'implantation en faisant appel à la méthode "calc_date_plus_tard" de la classe "node". Puis elle teste l'appartenance d'opérations distribuées sur FPGA au chemin critique en comparant les dates de début au plus tôt et les dates de début au plus tard. En effet, si ces deux dates sont égales alors l'opération appartient au chemin critique. Si ce test est positif alors la valeur test prend "True", sinon elle prend "False".

6.2.6 Creation_liste_sous_graphes

La fonction "creation_liste_sous_graphes" crée et retourne une liste "liste sg" des sous-graphes distribués sur composants reconfigurables. Cette fonction reçoit le graphe d'implémentation "graphe", et la liste "operations sur fpga" des opérations distribuées sur les différents composants reconfigurables.

La liste "liste sg" contient une liste des différents sous-graphes distribuées sur FPGA. Chaque sous-graphe est sous la forme d'une liste des opérations qui le constituent.

1. Un dictionnaire est un type de données puissant. Les dictionnaires sont des objets pouvant en contenir d'autres, à l'instar des listes. Cependant, au lieu d'héberger des informations dans un ordre précis, ils associent chaque objet contenu à une clé.

6.2.7 Creation_fichier_sdx_ic

La fonction "creation_fichier_sdx_ic" est utilisée pour créer un fichier sdx pour chaque sous-graphe distribué sur FPGA. Cette fonction utilise le fichier "fichier.sdx" spécifié par l'utilisateur, la liste de sous-graphes distribués sur FPGA "liste sg", la liste "liste operations" des opérations que peuvent exécuter les composants reconfigurables et la liste "liste fpga" des définitions des composants reconfigurables et les listes "actuators" et "sensors" contenant des informations à propos des données envoyées et reçus par les sous-graphes distribués sur FPGA.

La fonction "creation_fichier_sdx_ic" crée un nouveau fichier "fichiericN.sdx", où N correspond à l'indice du sous-graphe dans la liste "liste sg", pour chaque sous-graphe distribué sur FPGA. Dans ce fichier, la fonction "creation_fichier_sdx_ic" commence par définir les différents actionneurs et capteurs en utilisant les informations contenues dans les listes "actuators" et "sensors". Puis elle copie toutes les définitions de la partie "Algorithms" du fichier "fichier.sdx" sauf la définition de l'algorithme principal. A partir de la partie dépendance de l'algorithme principal, on identifie les capteurs et actionneurs correspondants au sous-graphe. Puis on crée des références aux différents éléments ("functions", "sensors", "actuators", ...) composant le sous-graphe ainsi que les connexions les reliant.

Ensuite, la fonction "creation_fichier_sdx_ic" définit l'opérateur FPGA et spécifie la durée d'exécution et la surface nécessaire pour chaque opération qu'il peut exécuter à partir des listes "liste opérations" et "liste fpga". Puis, on définit l'architecture principale contenant seulement l'opérateur FPGA défini précédemment. Enfin, on ajoute les noms de l'algorithme principal et de l'architecture principale et la contrainte temporelle qui est nulle.

6.2.8 Optimisation temporelle

L'optimisation temporelle est effectuée en lançant l'heuristique de SynDEx-IC. On utilise, comme pour lancer SynDEx (voir page 114) , la bibliothèque standard "os" de Python.

Pour chaque sous-graphe sur FPGA, on lance l'heuristique de SynDEx-IC qui sauvegarde la durée d'exécution obtenue après optimisation dans un fichier "resultat.txt". Les différentes durées d'exécution calculées par SynDEx-IC sont regroupées dans une liste "t" tel que l'indice de chaque durée dans cette liste est égal à l'indice du sous-graphe correspondant dans la liste "liste sg".

6.2.9 Graphe_update

La fonction "graphe_update" met à jour le graphe d'implémentation après l'optimisation temporelle. Elle produit le dictionnaire "graphe" modélisant le graphe d'implémentation mis à jour. Cette fonction reçoit en entrée le graphe d'implémentation "graphe", la liste "liste sg" des sous-graphes distribués sur FPGA et la liste "t" des nouvelles durées d'exécution.

Pour chaque sous-graphe distribué sur FPGA de la liste "liste sg", la fonction "graphe_update" remplace les opérations constituant le sous-graphe par une seule opération dont la durée d'exécution est la valeur contenue dans la liste "t" produite par SynDEx-IC. Elle spécifie aussi les successeurs et prédécesseurs pour le nœud représentant le sous-graphe. Elle indique le composant reconfigurable (opérateur dont le nom commence par

fpga) qui l'exécute. Cette fonction met à jours les listes "successeurs" et "predecesseurs" des opérations du graphe d'implémentation qui sont connectées au sous-graphe.

6.2.10 Creation_sdx_contraint

La fonction "creation_sdx_contraint" a pour but de créer un nouveau fichier "fichier_3.sdx" contenant l'algorithme de l'application, l'architecture cible et des contraintes de distribution qui garantissent de garder le même partitionnement des opérations sur les différents opérateurs. Elle reçoit en entrée le graphe d'implémentation mis à jour "graphe", la liste "liste sg" des sous-graphes distribués sur FPGA, le fichier "fichier_2.sdx" obtenu après transformation du graphe et les listes "sensors" et "actuators" contenant des informations sur les différents ports d'entrée et de sortie des différents sous-graphes.

Cette fonction commence par créer un dictionnaire regroupant les informations des ports d'entrée et de sortie pour chaque sous-graphe distribué sur FPGA. Puis, elle crée dans le fichier "fichier_3.sdx" les définitions des opérations qui représentent des sous-graphes distribués sur FPGA. Ensuite elle copie les définitions des opérations contenues dans le fichier "fichier_2.sdx" sauf l'algorithme principal qui est transformé en regroupant les opérations constituant chaque sous-graphe distribué sur FPGA en une seule opération et en adaptant les connexions entre ces opérations et leurs prédécesseurs et successeurs.

De la même façon, dans la partie Architecture, cette fonction copie les définitions des opérateurs et moyens de communication contenus dans le fichier "fichier_2.sdx". Puis cette fonction définit un opérateur pour chaque opération modélisant un sous-graphe distribué sur FPGA. Elle copie aussi l'architecture principale du fichier "fichier_2.sdx" en remplaçant les opérateurs dégénérés par ces opérateurs capables d'exécuter les opérations des sous-graphes distribués sur FPGA.

Enfin, cette fonction ajoute dans le fichier "fichier_3.sdx" les informations nécessaires pour contraindre SynDEx à garder la même distribution des opérations du fichier "fichier_2_adeq.sdx"

6.2.11 Calcul des dates au plus tôt

Après l'optimisation temporelle, il faut recalculer les dates de début et de fin des opérations du graphe d'algorithme. Pour effectuer cette opération, on utilise SynDEx en lui passant le fichier "fichier_3.sdx". Puisque ce fichier contient des contraintes sur le partitionnement des opérations sur les différents opérateurs de l'architecture, SynDEx calcule seulement les dates de début et de fin au plus tôt. Le résultat est obtenu dans le fichier "fichier_3_adeq.sdx".

6.2.12 Date_update

La fonction "date_update" permet de mettre à jour les dates du graphe d'implantation à partir du fichier "fichier_3_adeq.sdx". Elle reçoit en plus de ce fichier le dictionnaire "graphe" modélisant le graphe d'implémentation et la liste "liste sg" des sous-graphes distribués sur FPGA. Elle retourne le graphe avec les nouvelles dates et une liste "liste_refactorisation" des indices dans la liste des sous-graphes "liste sg" des sous-graphes à refactoriser.

A partir de la partie du fichier "fichier_3_adeq.sdx" contenant l'adéquation, cette fonction met à jour les dates de début et de fin au plus tôt des opérations du graphe

d'implémentation. En parallèle, elle calcule la longueur du nouveau chemin critique. Puis, elle calcule les dates de début et de fin au plus tard de chacun des nœuds du graphe en faisant appel à la méthode "calc_date_plus_tard" de la classe "node". Enfin, elle calcule les nouvelles flexibilités des sous-graphes distribués sur FPGA et ajoute les indices des sous-graphes dont la flexibilité n'est pas nulle à la liste "liste refactorisation".

6.2.13 Fichier_sdx_ic_update

Cette fonction modifie la contrainte temporelle contenue dans les fichiers sdx pour que SynDEx-IC refactorise les sous-graphes qui peuvent l'être sans allongement du chemin critique. Elle reçoit les différents fichiers sdx destinés à SynDEx-IC et créés par la fonction "creation_fichier_sdx_ic", la liste "liste sg" des sous-graphes distribués sur FPGA et la liste "liste refactorisation" des sous-graphes à refactoriser.

Cette fonction identifie les fichiers à modifier à partir des éléments de la liste "liste refactorisation" et le nom du sous-graphe à refactoriser à partir de la liste "liste sg". Ce nom de sous-graphe est utilisé pour accéder au nœud correspondant et pour pouvoir connaitre la durée d'exécution minimale et la flexibilité d'ordonnancement. Une fois toutes ces informations connues, la fonction "fichier_sdx_ic_update" modifie la dernière ligne du fichier sdx en indiquant la somme de la durée d'exécution minimale et la flexibilité d'ordonnancement comme contrainte temporelle.

6.2.14 Optimisation de surface et génération du code VHDL

Comme nous avons vu dans la section 4.8.2, l'optimisation de surface est effectuée en refactorisant les frontières de factorisation tant qu'elles ne sont pas sur le chemin critique. Cette opération est préparée par la fonction "fichier_sdx_ic_update" en modifiant les contraintes temporelles dans les fichiers sdx correspondants aux sous-graphes à refactoriser.

Pour générer le code VHDL des sous-graphes distribués sur FPGA, il suffit de lancer l'heuristique de SynDEx-IC qui refait l'adéquation, donc refactorise les sous-graphes dont les fichiers sdx ont été modifiés, puis génère des macros codes qui seront traduits en codes VHDL.

6.2.15 Generation_ip

La fonction "generation_ip" permet de générer les IP de communication nécessaires pour gérer la communication et la synchronisation entre chaque FPGA et les autres composants. Elle reçoit le fichier "fichier.sdx" spécifié par l'utilisateur, le dictionnaire "graphe" représentant le graphe d'implantation, la liste "liste ip com" des IP de communication à créer et les listes " sensors" et "actuators" contenant les informations à propos des données reçues et envoyées par les IP de communication.

Pour chaque IP de communication, cette fonction commence par établir une liste "data list" des données à transférer à partir des instructions d'allocation de mémoire dans le fichier m4 et des listes "actuator" et "sensor" établies par la fonction "creation_graphe_implantation". Chaque élément de cette liste est de la forme (nom référence productrice, nom port, taille). Puis, après avoir demandé à l'utilisateur la largeur du bus données (com_bus_width), à partir du premier thread (correspondant aux communications avec le composant distant) du fichier m4, pour chaque opération de communication,

on ajoute un élément au dictionnaire communication avec la clé "indice com". Cet "indice com" est un entier correspondant à l'indice de l'opération de communication dans la liste des communications à effectuer. Chaque élément de ce dictionnaire est de la forme "[nom référence productrice, nom port, liste références réceptrices, liste ports récepteurs, nombre paquets, sens, liste synchronisation]" où "sens" égale à '0' si c'est une réception et '1' si c'est un envoie et "liste synchronisation" est une liste des opérateurs dégénérés auxquels il faut envoyer un signal de synchronisation après avoir effectué cette communication. A ce stade, les éléments correspondants aux "liste références réceptrices", "liste ports récepteurs" et "liste synchronisation", sont remplis par des listes vides. A partir des instructions *send* du deuxième thread et des dépendances du graphe d'algorithme, on peut remplir les listes "liste référence réceptrice" et "liste port récepteur" pour les opérations de réception. Quant aux opérations d'envoi, ces listes restent vides car on n'aura pas besoin de ces informations. Puis à partir des éléments du dictionnaire "communication", on établit pour chaque élément la liste "liste synchronisation". On passe maintenant à la génération du VHDL. Un dossier nommé VHDL est créé dans le dossier contenant le fichier "fichier.sdx" spécifié par l'utilisateur. On y copie alors les fichiers VHDL des composants qui ne nécessitent pas de transformation. Puis, on calcule les différents paramètres (autre que com_bus_width) et on les intègre dans le fichier top level "com.vhd". Enfin, le fichier "rom.vhd" contenant la description de la mémoire est créé en utilisant les informations du dictionnaire communication.

6.3 Norme d'encodage vidéo H.264/AVC

Nous disposons maintenant d'un outil complet qui supporte les architectures mixtes. Après avoir validé son fonctionnement sur des exemples simples, nous allons le valider sur une application plus complexe : l'estimation de mouvement de la norme H.264/AVC.

La vidéo numérique exploite la persistance rétinienne qui résulte du temps de traitement biochimique des signaux optiques par la rétine et le cerveau. En effet, en affichant une succession d'images (Trames) à la fréquence minimale de 25 Trames par seconde, le cerveau humain l'aperçoit comme une vidéo fluide. Le stockage et la transmission de ces vidéos numériques nécessitent un espace mémoire et une bande passante énormes. Par exemple, la transmission d'une séquence couleur de résolution CIF (288 lignes, 352 colonnes), chaque pixel codé sur 24 bits (8 pour les niveaux de gris et 8 pour chacun des deux composants de couleur) avec une résolution temporelle de 25 images par seconde, il nous faut un débit de 60825600 bit/s ; ce qui correspond à une bonde passante de 58 Mbit/s. D'où la nécessité de la compression vidéo. La compression vidéo utilise deux types de prédictions : la prédiction intra qui exploite la redondance spatiale dans une même image et la prédiction inter qui exploite la redondance temporelle entre les images de la séquence. Dans cette section, nous allons nous intéresser à la norme de compression vidéo H.264/AVC (qui permet de réduire la bande passante de moitié par rapport à la norme précédente H.262) [ITU-T, 2003], apparue en 2003, et essentiellement à l'estimation de mouvement.

Les trames constituant une séquence vidéo, sont regroupées en “Groupes d'images" (GOP pour Group Of Pictures en anglais). Ces GOP sont encodés séparément les uns des autres. Chaque GOP contient un nombre déterminé d'images. Il commence par une image I codé entièrement par la prédiction intra et servant de référence pour la prédiction inter de l'image suivante.

L'encodeur de la norme H.264 est représenté par la figure 6.2.

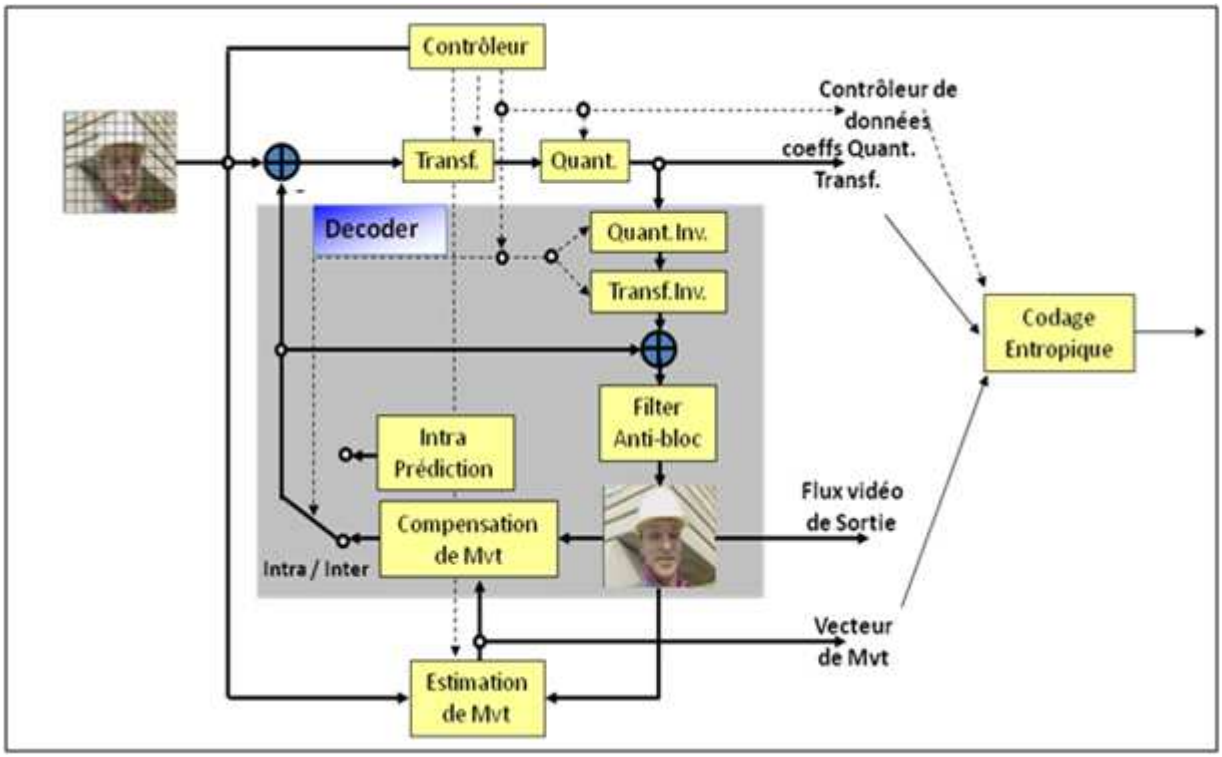

FIGURE 6.2 – Schéma bloc du encodeur H.264/AVC

Lors de l'encodage, chaque Image est découpée en macro-blocs (MB) de 16x16 pixels. Chaque MB courant (à encoder) passe par la prédiction intra ou inter pour avoir un MB prédit dont les valeurs des pixels seront déduites des valeurs des pixels du MB courant pour obtenir l'erreur de prédiction. Cette erreur de prédiction sera transformée, quantifiée puis codée en utilisant un codage entropique. Pour avoir les mêmes images de références pour la prédiction inter au niveau du codeur et du décodeur, l'erreur transformée et quantifiée subit la quantification inverse, la transformation inverse, puis est additionnée au MB prédit pour obtenir le MB reconstruit. Après reconstruction de tout les MB de l'image, celle ci est filtrée puis sauvegardée pour être utilisée comme référence dans la prédiction inter. Toutes ces étapes seront détaillées par la suite.

6.3.1 Prédiction intra

La prédiction intra exploite la redondance spatiale au sein de l'image même. En effet, elle prédit le MB courant en se basant sur les valeurs des pixels voisins selon une direction déterminée. La norme de codage vidéo H.264/AVC utilise deux types de prédiction intra : la prédiction intra 16x16 et la prédiction intra 4x4 [Chun-Ling Yang et al., 2004].

La prédiction intra 16x16 [Loukil et al., 2009] de la norme H.264/AVC peut être effectuée selon 4 modes représentés par la figure 6.3. Ce type de prédiction n'est efficace dans les zones homogènes de l'image puisque il prédit les valeurs de tous les pixels du MB en utilisant le même mode.

La prédiction intra 4x4 consiste à découper le MB en 16 blocs 4x4 et de prédire chacun de ces blocs selon un des 9 modes représentés dans la figure 6.4. Ce type de prédiction permet de mieux prédire les blocs des zones avec des textures.

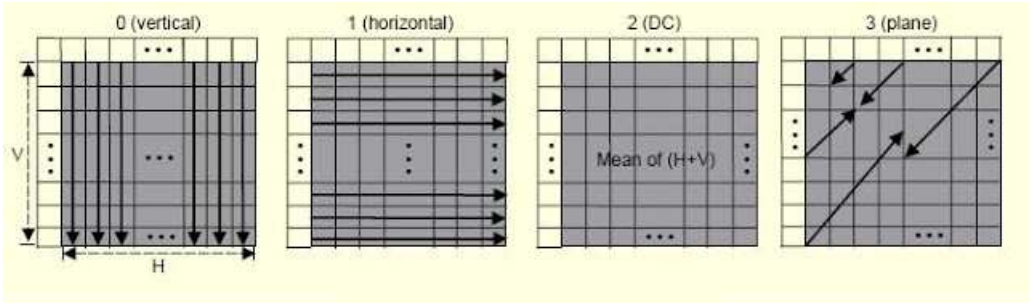

FIGURE 6.3 – Modes de prédiction intra 16x16 [Richardson, 2003]

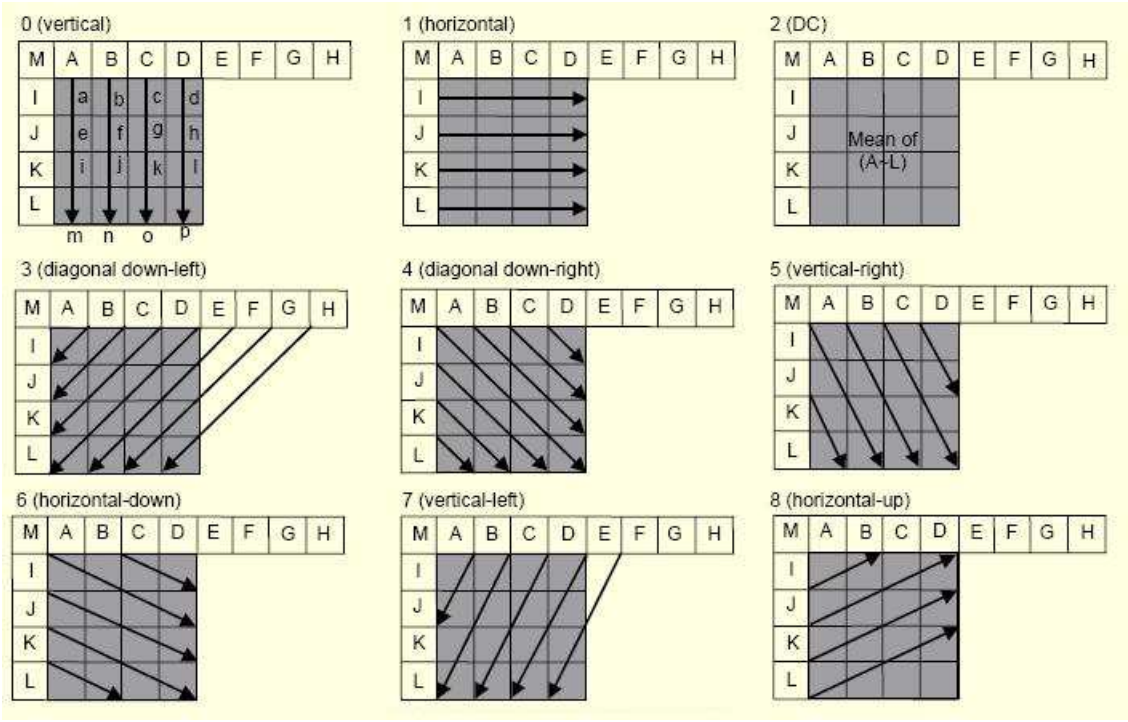

FIGURE 6.4 – Modes de prédiction intra 4x4 [Richardson, 2003]

6.3.2 Prédiction inter

La prédiction inter exploite la redondance temporelle entre les images successives de la séquence vidéo [Richardson, 2003]. Elle consiste à associer au bloc courant le bloc lui ressemblant le plus dans l'image référence reconstruite (encodée puis décodée) et un vecteur de mouvement. Le bloc le plus ressemblant dans l'image référence est le bloc prédit et la différence des positions respectives de ces blocs dans les images est le vecteur de mouvement. Cette prédiction permet d'effectuer un bon taux de compression. En effet, les blocs courant et prédit se ressemblent ; donc l'erreur de prédiction est faible. Cette efficacité vient au dépend d'une complexité élevée de l'estimation de mouvement. Pour réduire la complexité de l'estimation de mouvement, plusieurs travaux de recherche ont été effectués. Les résultats de ces recherches seront exposés dans la section suivante.

6.3.3 Transformées

La norme H.264 utilise une transformée en cosinus entière qui s'applique sur les blocs 4x4 [Gouyet and Sablier, 2007]. Cette transformée, comme son nom l'indique, ne manipule que des entiers. Ainsi on évite les distorsions de la transformée inverse. De plus, cette transformée n'utilise que des additions et des décalages ; ce qui facilite son implémentation.

En plus de cette transformée en cosinus entière, le codeur H.264 utilise la transformée HADAMARD pour les coefficients DC de chaque bloc résiduel d'une prédiction INTRA 16x16 et les coefficients de la chrominance.

Ces transformées permettent regrouper les coefficients nuls pour augmenter l'efficacité du codage entropique.

6.3.4 Quantification

La quantification est responsable de la perte de l'information peu pertinente. En effet, cette étape annule les coefficients à faible énergie.

La norme H.264 fixe 52 seuils de quantification avec une augmentation de 12.5% au passage d'un seuil de quantification au suivant. L'augmentation de ce seuil de quantification augmente l'efficacité de la compression vidéo puisque le nombre de coefficients qui s'annulent augmente. Mais en contre partie, la qualité de la vidéo décodée diminue.

6.3.5 Codage entropique

Le codage entropique élabore la séquence binaire (bitstream) contenant toutes les informations pertinentes et pouvant être décodées par le décodeur H.264 pour obtenir la vidéo décompressée [Gouyet and Sablier, 2007].

Deux types de données sont concernées par ce codage entropique : les coefficients des blocs résiduels transformés et quantifiés et les éléments syntaxiques concernant l'image, le bloc, la nature de prédiction ....

Ces deux types de données peuvent être codés (selon le profil du codage H.264) soit avec le même code "CABAC" (Context-based Adaptative Binary Arithmetic Code) soit en utilisant un code adaptatif "CAVLC" (Context-based Adaptative Variable Length Coding) [Damak et al., 2010] pour les coefficients transformés quantifiés et un code "Exp-Golomb" pour les éléments syntaxiques.

6.3.6 Filtrage anti-blocs

Le traitement des images en blocs produit des artéfacts visibles aux bord des ces blocs. La figure 6.5 (a) présente une image avec des artéfacts de blocs visibles. Pour remédier à ce problème, la norme H.264/AVC définit un filtre de "déblocage" adaptatif en boucle (loop filter)[Damak et al., 2011]. La rigueur du filtrage dépend de la position du bloc, de la nature de la prédiction utilisée et du seuil de quantification utilisé [List et al., 2003]. Après filtrage, la pixellisation est alors réduite et la qualité subjective est considérablement améliorée comme le montre la figure 6.5 (b).

6.4 Estimation de mouvement

L'estimation de mouvement est une technique importante pour la compression vidéo. Elle permet d'exploiter la redondance temporelle entre les images successives de la sé-

(a) (b)

FIGURE 6.5 – Effet du filtre anti-blocs : (a) image non filtrée, (b) image filtrée

quence vidéo. Cette technique consiste à associer à chaque bloc de l'image courante un vecteur de mouvement qui correspond à la différence de position de ce bloc courant par rapport au bloc qui lui ressemble dans une image référence. La compression est d'autant plus efficace que les deux blocs se ressemblent plus. En effet, plus les blocs (courant et prédit) se ressemblent, moins sera l'énergie de l'erreur de prédiction à transmettre. Mais chercher le bloc prédit dans l'image référence nécessite une capacité de traitement et un temps de calcul énormes. Pour réduire ce temps de traitement, plusieurs travaux de recherche ont été effectués. Ces travaux touchent plusieurs aspects de l'estimation de mouvement : taille de la fenêtre de recherche, choix du centre de la fenêtre de recherche et l'algorithme de correspondance de blocs (bloc matching) utilisé. Dans la suite de cette section nous allons présenter quelques uns de ces travaux.

6.4.1 Centre de la fenêtre de recherche

La détermination du centre de la fenêtre de recherche est important pour réduire les positions testées avant de trouver le bloc le plus ressemblant au bloc courant. Gallant et all. [Gallant and Kossentini, 1998] partent du fait que les blocs voisins effectuent un déplacement proche les uns des autres. Ils choisissent alors comme centre de la fenêtre de recherche la position correspondante à un déplacement d'un vecteur de mouvement donné par la médiane des vecteurs de mouvement des blocs voisins comme le montre la figure 6.6.

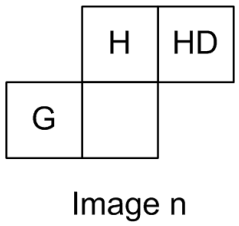

FIGURE 6.6 – Blocs pour trouver le centre de recherche selon [Gallant and Kossentini, 1998]

Werda et all. [Werda et al., 2007] se sont inspirés de l'approche précédente pour proposer une nouvelle méthode d'estimation du centre de la fenêtre de recherche. Cette méthode

se base sur la comparaison, en utilisant le caractère de comparaison SAD, du bloc courant avec le bloc qui occupe la même position dans l'image référence et les blocs de l'image de référence qui correspondent à un déplacement correspondant à un déplacement de vecteur de mouvement de chacun des voisins du bloc courant comme le montre la figure 6.7. Cette méthode est mieux adaptée à l'implémentation logicielle car elle réduit les structures conditionnelles qui causent une rupture du pipeline. De plus, cette méthode produit une vidéo reconstruite de qualité acceptable [Werda, 2011].

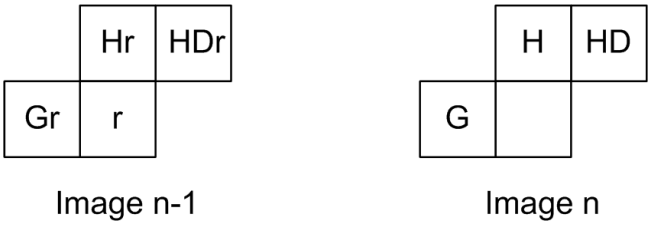

FIGURE 6.7 – Blocs pour trouver le centre de recherche selon [Werda et al., 2007]

6.4.2 Taille de la fenêtre de recherche

Après avoir estimé avec plus de précision le centre de la fenêtre de recherche, nous pouvons réduire la taille de la fenêtre de recherche. Chaouch et all. [Chaouch et al., 2007] ont effectué une étude statistique des vecteurs de mouvement sur plusieurs séquences de test. Cette étude a permis de fixer une taille de fenêtre de recherche de $[-9, 9]$ sur l'axe des abscisses et de $[-7, 7]$ sur l'axe des ordonnées. Ainsi, les mouvement horizontaux sont favorisés par rapport aux mouvements verticaux ; ce qui correspond à la nature des séquences vidéos. Cette réduction de la taille de la fenêtre de recherche permet une réduction de 25% de la complexité de l'estimation de mouvement contre une légère baisse de la qualité vidéo.

6.4.3 Algorithmes d'estimation de mouvement

La recherche du bloc ressemblant au bloc courant en testant toutes les positions dans une fenêtre de recherche donne le meilleur résultat. Mais cette méthode (Full search [Richardson, 2002]) nécessite une intensité de calcul élevé qui rend l'implémentation temps réel difficile à atteindre. Pour réduire cette demande en capacité de calcul, plusieurs travaux ont proposé des algorithmes rapides d'estimation de mouvement qui réduisent le nombre de positions à tester. De cette façon, on peut réduire le temps de recherche mais il est probable de tomber sur un minimum local. Malgré cette possible perte de compression, ces algorithmes sont fréquemment utilisés dans les codeurs vidéo car ils permettent une nette réduction de la complexité et une forte accélération de l'estimation de mouvement. Nous allons présenter, dans la suite de ce paragraphe, quelques uns de ces algorithmes d'estimation rapide de mouvement à savoir : Three Step Search [Liou, 1994], Diamond Search [Zhu and Ma, 2000], Hexagone-Based Search [Ce Zhu et al., 2002], Horizontal Diamond Search [Ben Ayed et al., 2007] et Line Diamond Parallel Search [Werda et al., 2010].

Three Step Search

L'algorithme Three Step Search [Liou, 1994] se base, comme son nom l'indique, sur trois étapes illustrées dans la figure 6.8. La première étape consiste à tester la position

centrale, les quatre positions distantes de quatre pixels (verticalement et horizontalement) par rapport à cette position centrale et les quatre sommets du carré dont le centre est cette position centrale et passant par les autres positions testées. La position donnant le plus de ressemblance lors de la première étape est considérée le centre de la seconde étape. Lors de la deuxième étape, le même motif est testé mais en réduisant de moitié la distance entre les positions testées. De la même façon, la position donnant le plus de ressemblance est considérée le centre de la troisième étape. Lors de cette troisième étape, la position centrale est comparée à ces huit voisins.

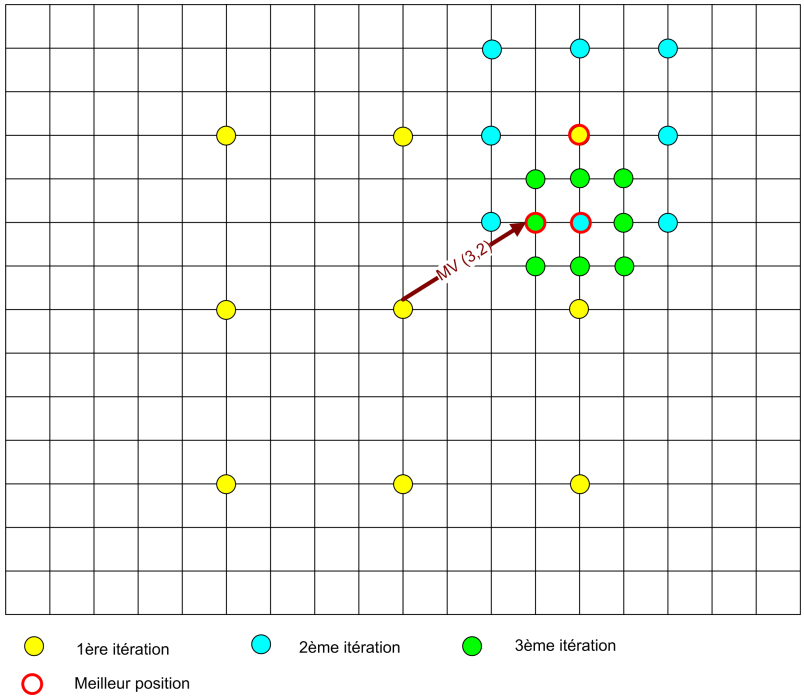

FIGURE 6.8 – Etapes de l'algorithme Three Step Search

Diamond Search

La figure 6.9 représente les étapes d'exécution de l'algorithme Diamond Search [Zhu and Ma, 2000]. Cet algorithme teste les neufs positions formées par le centre de la fenêtre de recherche et les huit positions formant un losange dont les sommets sont à une distance de deux pixels par rapport au centre (points jaunes de la figure 6.9). A chaque itération, ce même motif est utilisé en prenant pour centre la position donnant le plus de ressemblance de l'itération précédente. Si deux itérations successives donnent le même centre, on passe à une étape de raffinement en testant le centre et les quatre positions adjacentes verticalement et horizontalement.

Hexagone-Based Search

L'algorithme Hexagone-Based Search [Ce Zhu et al., 2002] est semblable au Diamond Search. La seule différence entre ces deux algorithmes est le modèle de recherche. En effet, le Diamond Search utilise un motif en losange, alors que le Hexagone-Based Search se base sur un motif hexagonal (motif en jaune dans la figure 6.10). Ainsi la position centrale et les six positions formant l'hexagone tout autour sont testées. Puis, à chaque

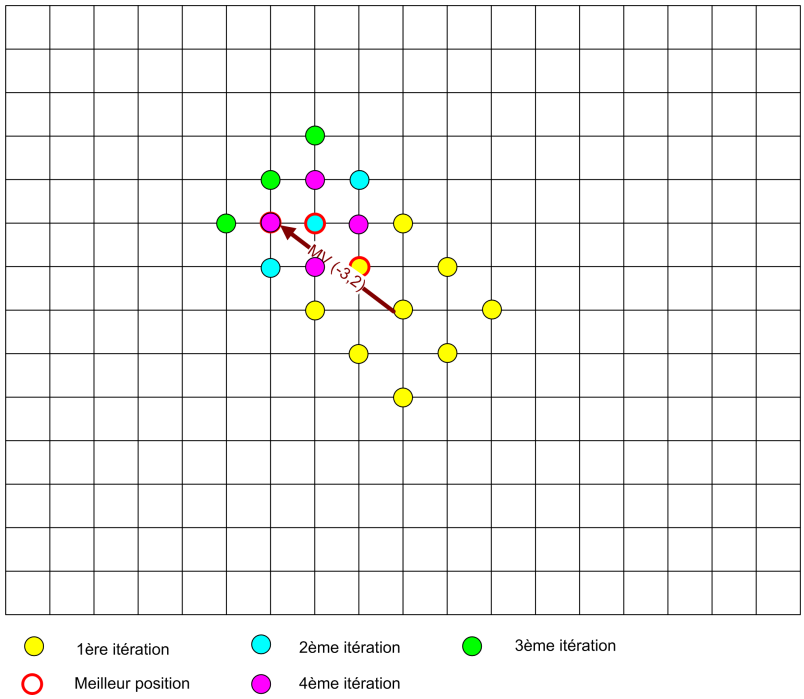

FIGURE 6.9 – Etapes de l'algorithme Diamond Search

itération, on prend la position donnant le plus de ressemblance pour centre et on réutilise le même motif jusqu'à ce que deux itérations successives donnent le même centre. Enfin, on teste la position centrale et les quatre positions qui lui sont adjacentes verticalement et horizontalement.

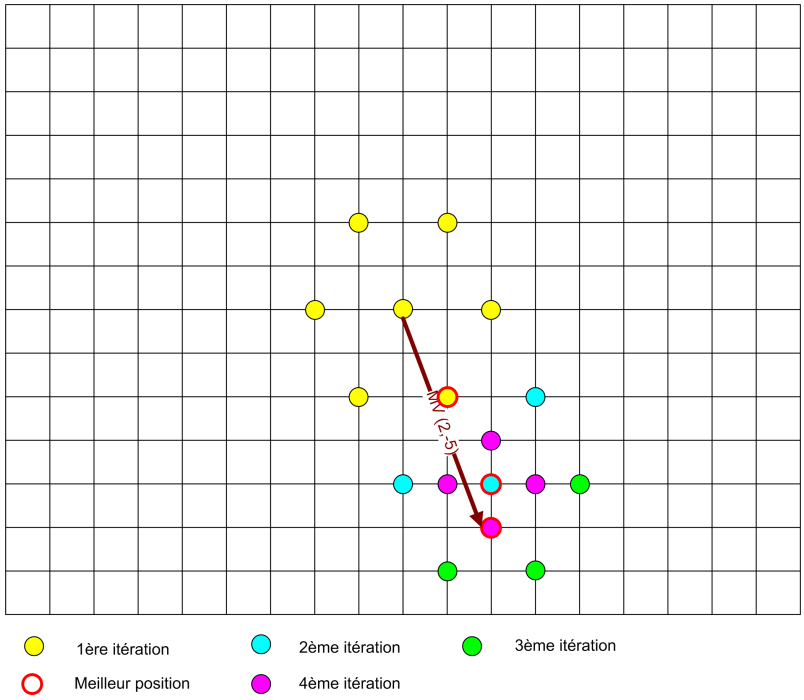

FIGURE 6.10 – Etapes de l'algorithme Hexagone-Based Search

Horizontal Diamond Search

L'algorithme Horizontal Diamond Search [Ben Ayed et al., 2007] favorise la recherche selon la composante horizontale du mouvement. Il utilise le motif représenté en jaune dans la figure 6.11. A chaque itération, les positions formant ce motif sont testées en prenant comme centre la position donnant le plus de ressemblance de l'itération précédente. La recherche s'arrête lorsqu'on obtient le même centre pour deux itérations successives.

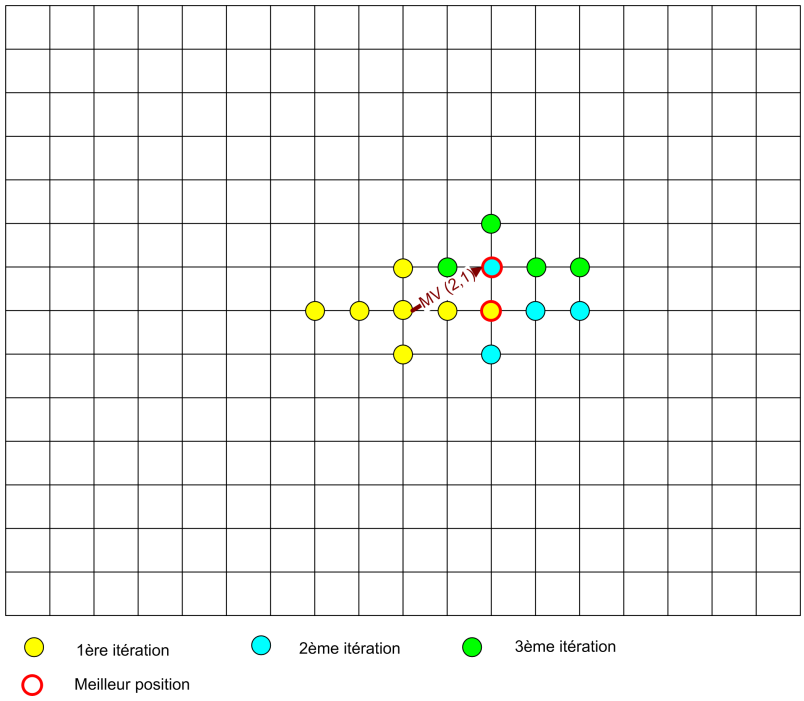

FIGURE 6.11 – Etapes de l'algorithme Horizontal Diamond Search

Line Diamond Parallel Search

L'algorithme Line Diamond Parallel Search [Werda et al., 2010] se déroule en répétant deux étapes. La première étape est le choix de la direction de recherche. Elle s'effectue en comparant les positions du centre de la fenêtre de recherche et les quatre position adjacentes verticalement et horizontalement. La direction choisie est celle de la position la plus ressemblante. On passe alors à l'étape de recherche dans la direction choisie en testant trois positions distantes chacune par rapport à l'autre de deux pixels dans la direction choisie. La position donnant le plus de ressemblance est utilisée comme centre pour le choix de direction suivante. On itère ces deux étapes jusqu'à ce que lors de l'étape de choix de direction le plus de ressemblance coïncide avec la position centrale (figure 6.12).

6.5 Implémentation de l'estimation de mouvement

Nous avons choisi d'implémenter l'algorithme d'estimation de mouvement Line Diamond Parellel Search car cet algorithme s'adapte à la nature du mouvement des séquences vidéos. En effet, cet algorithme favorise la recherche selon les directions horizontale et ver-

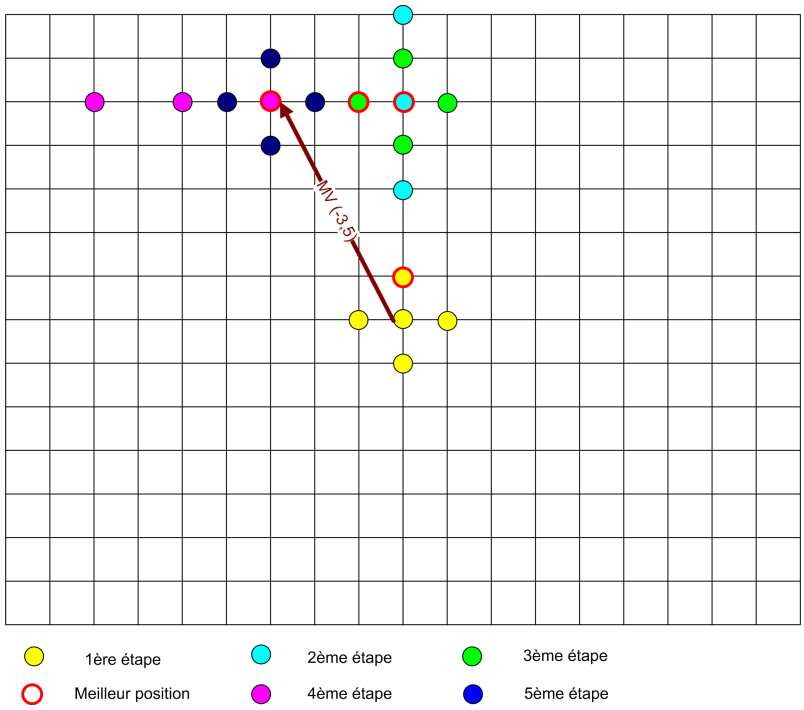

FIGURE 6.12 – Etapes de l'algorithme Line Diamond Parellel Search

ticale plutôt que la recherche selon une direction diagonale. Cet estimateur de mouvement est détaillé dans l'algorithme 5.

Cet algorithme reçoit en entrée le macrobloc courant contenant 256 pixels (16x16) et la fenêtre de recherche permettant un déplacement maximal de [-9,9] selon l'axe des abscisses et [-7,7] selon l'axe des ordonnées [Werda et al., 2007]. Cette fenêtre de recherche comporte 1020 pixels (34x30). Le résultat de l'exécution de cet algorithme est un vecteur de mouvement donnant le déplacement du macrobloc courant par rapport au macrobloc lui ressemblant de l'image référence.

L'algorithme LDPS commence par calculer les SAD de la position centrale et des quatre positions adjacentes (haut, bas, gauche et droite) par rapport au macrobloc courant.

Puis tant que SAD_c de la position centrale est supérieure au minimum des SAD des positions adjacentes, il teste trois positions (D1, D2 et D3) dans la direction donnant le minimum de SAD (lignes 6-8 de l'algorithme 5). La position donnant le minimum de SAD de ces trois positions testées (D1, D2 et D3) est considérée comme position centrale. Puis il calcule les SAD des positions adjacentes (haut, bas, gauche et droite) par rapport au macrobloc courant et compare le minimum d'entre eux avec la valeur SAD_c de la position centrale.

Lorsque la valeur minimale de $(min(SAD_c, SAD_h, SAD_b, SAD_g, SAD_d))$ est égale à SAD_c, le vecteur de mouvement est calculé en effectuant la différence entre la dernière position centrale et la position du macrobloc courant.

6.5.1 Graphe d'algorithme

Le graphe d'algorithme spécifié à SynDEx pour implémenter l'algorithme Line Diamond Parallel Search est donné dans la figure 6.13. Dans cette figure, on a coloré quelques connexions entre les différents sommets pour des raisons de clarté. Ce graphe d'algorithme contient des références à deux capteurs (sensor), six fonctions (function), un retard (de-

Algorithm 5 Algorithme de l'estimateur de mouvement Line Diamond Parellel Search

Require: Macrobloc courant MB et fenêtre de recherche
Ensure: Vecteur de mouvement V_m

for all $i \in (c, h, b, g, d)$ **do**
  $SAD_i = calc\_SAD(MB, P_i)$
end for
$min\_SAD = min(SAD_i)/i \in (h, b, g, d)$
while $SAD_c > min\_SAD$ **do**
  for all $i \in (D1, D2, D3)$ **do**
    $SAD_i = calc\_SAD(MB, P_i)$
  end for
  $SAD_c = min(SAD_{D1}, SAD_{D2}, SAD_{D3})$
  for all $i \in (h, b, g, d)$ **do**
    $SAD_i = calc\_SAD(MB, P_i)$
  end for
  $min\_SAD = min(SAD_i)/i \in (h, b, g, d)$
end while
$V_m = position(c) - position(MB)$

lay), six constantes et un actionneur (actuator). Comme SynDEx ne tolère pas les espaces dans les noms des définitions et des références, nous avons choisi d'écrire la première lettre de chaque mot en majuscule. Toutes les définitions utilisées seront détaillées dans la suite de ce paragraphe.

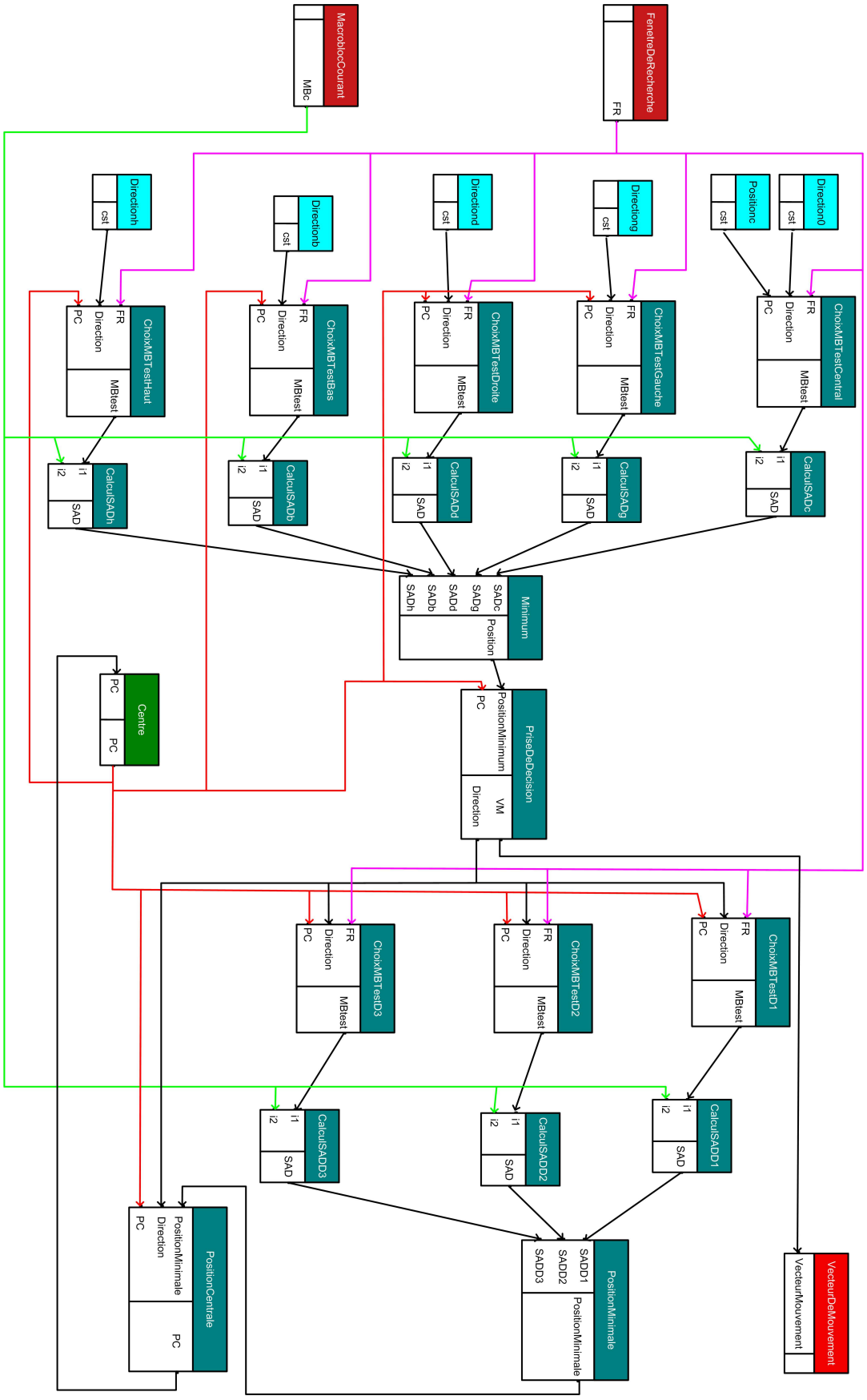

FIGURE 6.13 – Graphe d'algorithme du Line Diamond Parellel Search

Capteur FenetreRecherche

Chaque élément de la sortie du capteur "FenetreRecherche" représente un pixel de la fenêtre de recherche. Comme expliqué dans la figure 6.14, cette fenêtre de recherche comporte un macrobloc central (16x16 pixels) et les pixels représentant un déplacement maximal de [-9,9] selon l'axe des abscisses et [-7,7] selon l'axe des ordonnées. La taille de la fenêtre de recherche en nombre de pixels est alors de $(16+2\times9)\times(16+2\times7)=1020$. Ce capteur est référencé une seule fois dans le graphe d'algorithme. Le nom de cette référence est "FenetreDeRecherche" (en haut à gauche de la figure). Ce capteur a donc un seul port de sortie "FR" de 1020 caractères non signés (uchar).

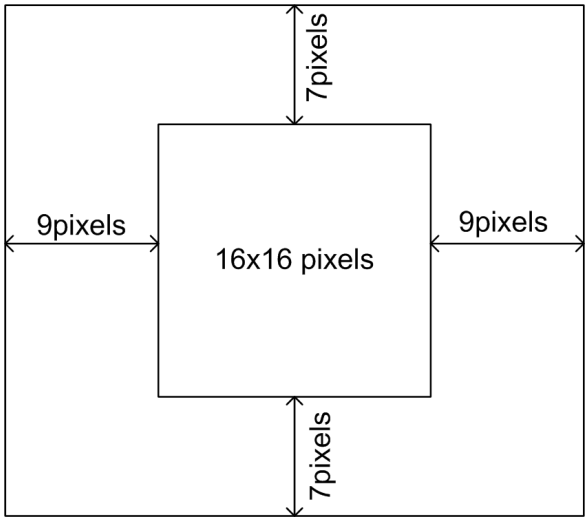

FIGURE 6.14 – Taille de la fenêtre de recherche

Capteur MBCourant

Chaque élément du port de sortie du capteur "MBCourant" représente un pixel du macrobloc courant (à encoder). Ce capteur est référencé une seule fois dans le graphe d'algorithme et le nom de cette référence est "MacroblocCourant". Ce capteur a un seul port de sortie "MBc" de 256 caractères non signés (uchar).

Constante ConstanteDoubletFloat

Cette définition de constante produit, comme l'indique son nom, produit un doublet de type float dans son unique port de sortie "cst". Cette constante est référencée une seule fois sous le nom "Positionc" pour initialiser la position centrale de la fenêtre de recherche.

Constante ConstanteInt

Cette définition de constante produit un entier dans son port de sortie "cst". Elle est référencée cinq fois dans le graphe d'algorithme. Ces références, "Direction0", "Directiong", "Directiond", "Directionb" et "Directionh" sont utilisées pour indiquer la direction de déplacement par rapport à la position centrale pour extraire respectivement le macrobloc test central, gauche, droite, bas et haut.

Fonction ChoixPosition

ChoixPosition est une définition de fonction qui a pour rôle d'extraire un macrobloc de test de la fenêtre de recherche. Elle a trois ports d'entrées : "FR" (1020 uchar) qui reçoit la fenêtre de recherche, "Direction" (1 int) pour indiquer la direction de déplacement par rapport à la position centrale et "PC" (2 float) pour indiquer les coordonnées de la position centrale. Cette définition possède un seul port de sortie "MBtest" (256 uchar) représentant le macrobloc test.

Le graphe d'algorithme fait appel à huit références de la fonction ChoixPosition : "ChoixMBTestCentral" pour extraire le MB test de la position centrale, "ChoixMBTest-Gauche" pour extraire le MB test de la position gauche par rapport à la position centrale, "ChoixMBTestDroite" pour extraire le MB test de la position droite par rapport à la position centrale, "ChoixMBTestBas" pour extraire le MB test de la position en dessous de la position centrale, "ChoixMBTestHaut" pour extraire le MB test de la position au dessus de la position centrale et "ChoixMBTestD1" "ChoixMBTestD2" et "ChoixMBTestD3" pour extraire les trois macroblocs tests de recherche dans la direction choisie.

Fonction Min5

La fonction Min5 cherche le minimum parmi cinq entiers et retourne le numéro de l'entrée correspondante à ce minimum. Elle reçoit cinq entrées : "SADc" (1 int), "SADg" (1 int), "SADd" (1 int), "SADh" (1 int) et "SADd" (1 int). Elle produit une seule sortie "position" (1 int) qui peut prendre une valeur comprise entre 0 et 4.

La fonction Min5 est référencée une seule fois dans le graphe d'algorithme sous le nom "Minimum" pour comparer les valeurs de SAD centrale par rapport aux quatre positions adjacentes.

Fonction PriseDecision

La fonction PriseDecision a pour but de verifier la condition de la boucle "tant que" de la ligne 5 de l'algorithme 5. Ainsi si cette condition est vérifiée, elle produit un entier indiquant la direction de recherche à suivre. Sinon, elle calcule le vecteur de mouvement. Cette fonction reçoit deux entrées : "PositionMinimum" (1 int) indiquant la position donnant le minimum de SAD et "PC" (2 float) indiquant les coordonnées de la position centrale. Elle possède deux ports de sortie : "VM" (2 float) représentant le vecteur de mouvement et "Direction" (1 int) représentant la direction choisie pour effectuer la recherche.

Cette fonction est référencée une seule fois dans le graphe d'algorithme sous le nom "PriseDeDecision" pour prendre la décision d'arrêter la recherche et de calculer le vecteur de mouvement ou de choisir une direction selon laquelle continuer la recherche.

Fonction Min3

La fonction Min3 cherche le minimum parmi trois entiers et retourne le numéro de l'entrée correspondante à ce minimum. Elle reçoit trois entrées : "SADD1" (1 int), "SADD2" (1 int) et "SADD3" (1 int). Elle produit une seule sortie "PositionMinimale" (1 int) qui peut prendre une valeur comprise entre 1 et 3.

La fonction Min3 est référencée une seule fois dans le graphe d'algorithme sous le nom "PositionMinimale" pour comparer les valeurs de SAD des trois positions de recherche selon la direction choisie et en choisir la nouvelle position centrale.

Fonction CalculPositionCentrale

La fonction CalculPositionCentrale calcule les coordonnées de la position centrale selon un déplacement donné dans la direction choisie. Elle reçoit trois entrées : "PositionMinimale" (1 int), "Direction" (1 int) et "PC" (2 float). Elle produit une seule sortie "PC" (2 float) qui représente les nouvelles coordonnées de la position centrale.

La fonction CalculPositionCentrale est référencée une seule fois dans le graphe d'algorithme sous le nom "PositionCentrale" pour calculer la nouvelle position centrale suite à la recherche selon la direction choisie.

Fonction AbsoluteDifference

La fonction AbsoluteDifference calcule la valeur absolue de la différence entre deux caractères non signées (uchar). Elle reçoit deux entrées identiques "i1" et "i2" (1 uchar) représentant chacune un pixel. Le seul port de sortie de cette fonction "resultat" (1 int) représente la valeur absolue de la différence entre les deux entrées.

La fonction AbsoluteDifference est référencée dans la fonction CalcSAD sous le nom "AbsDiff". Cette référence s'exécute 256 fois à chaque exécution de la fonction CalcSAD.

Fonction Accumulation

La fonction accumulation calcule la somme de 256 entiers. Elle reçoit dans son unique port d'entrée "input" (256 int) 256 entiers et produit dans son port de sortie "output" (1 int) un entier.

La fonction Accumulation est référencée dans la fonction CalcSAD sous le nom "Acc".

Fonction CalcSAD

La fonction CalcSAD calcule le SAD (Sum of Absolute Difference donné par l'équation 6.1 avec MBcour : macro-bloc courant et MBref : macro-bloc de référence. Cette définition de fonction possède deux ports d'entrée identiques "i1" et "i2" (256 uchar) représentant respectivement le macrobloc test et le macrobloc courant. Le seul port de sortie de cette fonction "SAD" (1 int) représente la valeur de SAD entre les deux macroblocs.

$$SAD = \sum_{i=1}^{16} \sum_{j=1}^{16} |MBcour(i,j) - MBref(i,j)| \tag{6.1}$$

Comme le montre l'équation 6.1, cette fonction comporte une répétition de 256 valeurs absolues de différence de deux pixels indépendants et une accumulation de ces valeurs.

Cette fonction est référencée huit fois dans le graphe d'algorithme : "CalcSADc" pour le calcule du SAD par rapport à la position centrale initiale, "CalcSADg" "CalcSADd" CalcSADh" et "CalcSADb" pour calculer les SAD respectivement des positions à gauche, à droite, au dessus et en dessous par rapport à la position centrale et "CalcSADD1" "CalcSADD2" et "CalcSADD3" pour calculer les SAD des trois positions testées lors de la recherche dans la direction choisie.

Retard PositionCentre

Cette définition de retard (Delay) mémorise un doublet de float. Donc elle dispose d'un port d'entrée et un port de sortie "PC" (2 float).

Cette définition de retard est référencée une seule fois, sous le nom "Centre", pour l'enregistrement de la nouvelle position centrale choisie après la recherche selon une direction.

Actionneur VecteurMouvement

Cet actionneur possède un seul port d'entrée "VecteurMouvement" (2 float). Chaque élément de cette entrée représente une coordonnée du vecteur de mouvement.

Cet actionneur est référencé une seule fois dans le graphe d'algorithme sous le nom "VecteurDeMouvement" pour donner les coordonnées du vecteur de mouvement.

6.5.2 Graphe d'architecture

L'architecture ciblée se base sur le FPGA Stratix EP3SL150F1152C2 [Altera, 2013b]. Cet FPGA comporte 113600 éléments logiques, 687 Kilo Octets de mémoire, 384 blocs DSP et 8 PLLs. Nous avons choisi d'implémenter un processeur NIOS II dans cet FPGA (figure 6.15). Par conséquent, l'architecture cible est formée par le processeur NIOS II et le reste du FPGA qui le contient. La communication entre ce processseur et les différents composants implémentés sur FPGA s'effectue à travers le bus Avalon [Altera, 2014a].

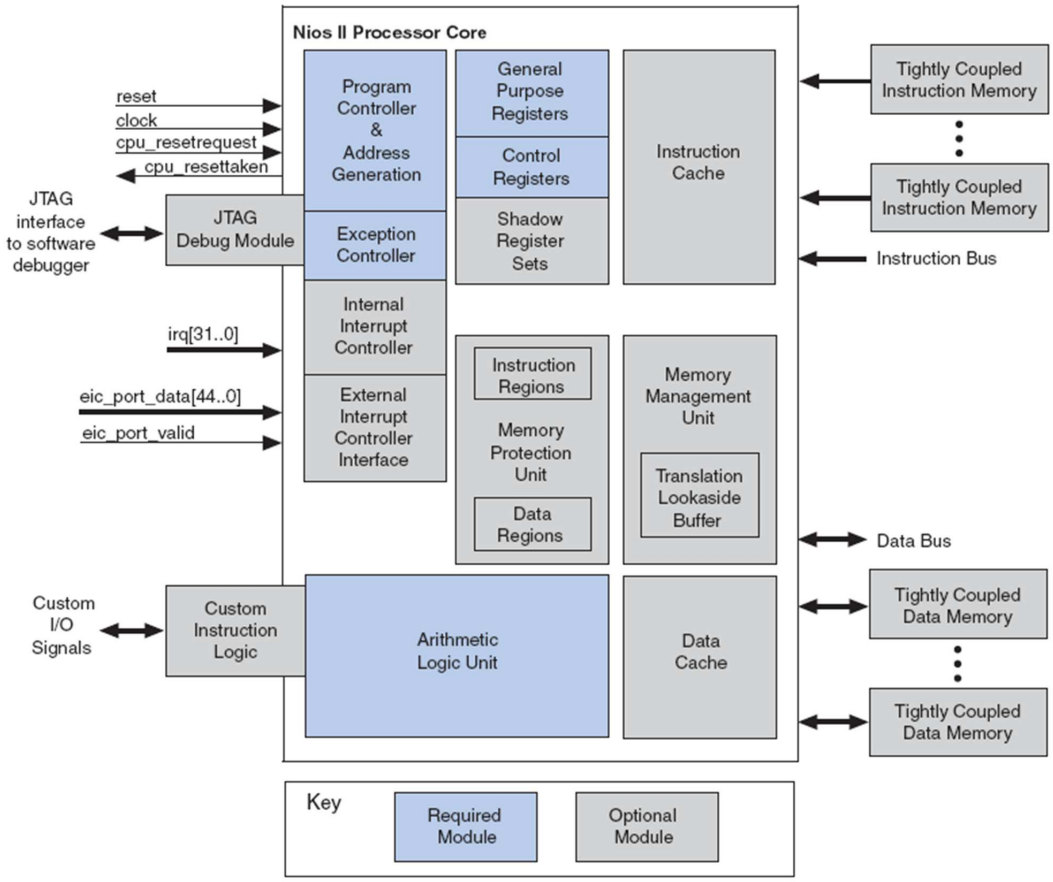

FIGURE 6.15 – Schéma bloc du processeur NIOS II

Le processeur NIOS II est un processeur softcore proposé par Altera. Il peut facilement être interfacé avec les différents périphériques que contiennent les cartes de développement

Altera. Le processeur NIOS II (figure 6.15) est un processeur RISC basé sur l'architecture Harvard, c'est à dire qu'il possède un bus instructions et un bus données séparées. Il possède au maximum 6 niveaux de pipeline cadencés à 50 MHz avec un bus de 32 bits. Les performances de ce processeur sont de 30 à 80 MIPS (*Million Instruction per Second*). Il existe 3 types de configurations du processeur NIOS II : configuration rapide (Fast) performante mais gourmande en ressources matérielles, configuration économique (economy) qui ne nécessite pas beaucoup de ressources matérielle mais peu performante et une configuration standard qui représente un bon compromis entre performances et utilisation de ressources.

Pour implémenter notre système, nous utilisons l'outil QuartusII d'Altera qui permet de gérer le flot de conception. Cet outil intègre l'environnement SOPC Builder, dont l'interface est présentée dans la figure 6.16, qui permet de créer un système sur puce (SOC). Ces systèmes peuvent intégrer plusieurs éléments comme des processeurs NIOS II, des mémoires, des contrôleurs de périphériques et d'autres composants personnalisés décrites en langage de description matériel.

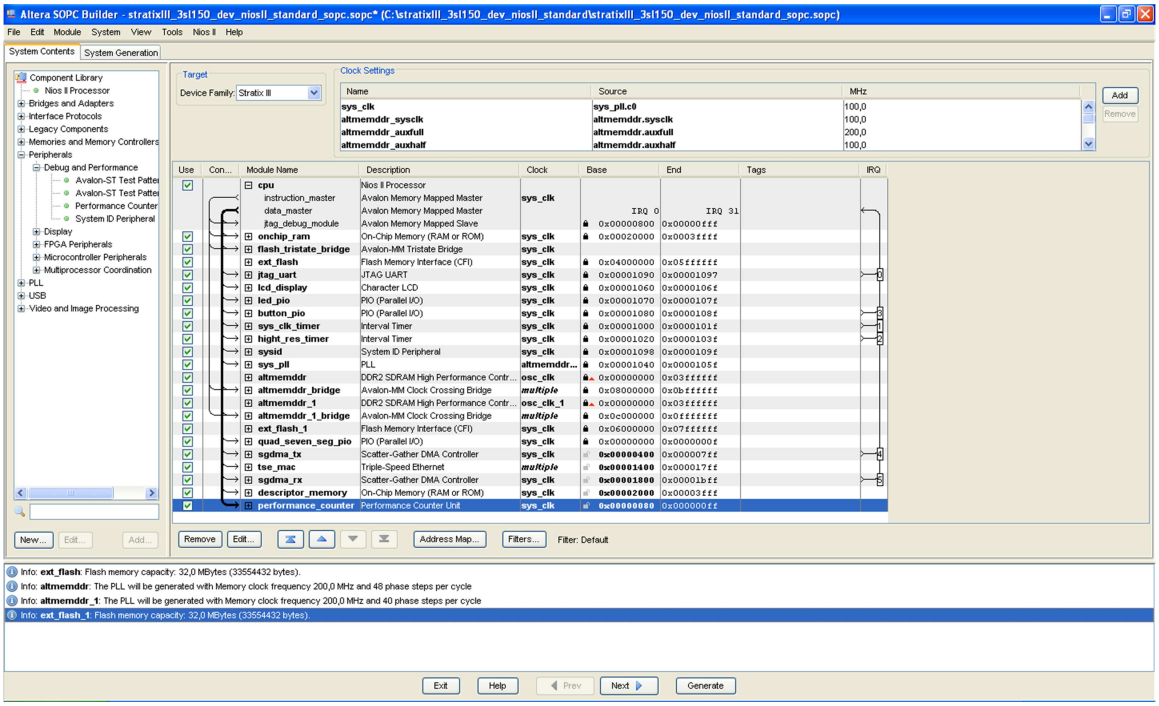

FIGURE 6.16 – Interface graphique de l'outil SOPC Builder

Après avoir créer la description matérielle du système, on peut utiliser l'outil NIOS II IDE (figure 6.17) pour programmer les différents processeurs NIOS II que contient notre système.

L'architecture que nous ciblons est constituée par un processeur NIOS II (fast) et un FPGA Stratix EP3SL150F1152C2. Ces deux composants sont connectés à travers le Bus Avalon. Le processeur NIOS II occupe 3001 éléments logiques, ce qui représente 3% de la surface totale du FPGA. Le graphe d'architecture que nous spécifions dans l'interface de SynDEx est 6.18. Les opérations d'entrées/sorties ne peuvent être exécutées que par le processeur NIOS II.

Pour pouvoir spécifier les durées d'exécution des différentes opérations sur le processeur NIOS II, on utilise le composant "Performance Counter" [Altera, 2011] d'Altera. Les résultats de ce profilage sont donnés dans la figure 6.19.

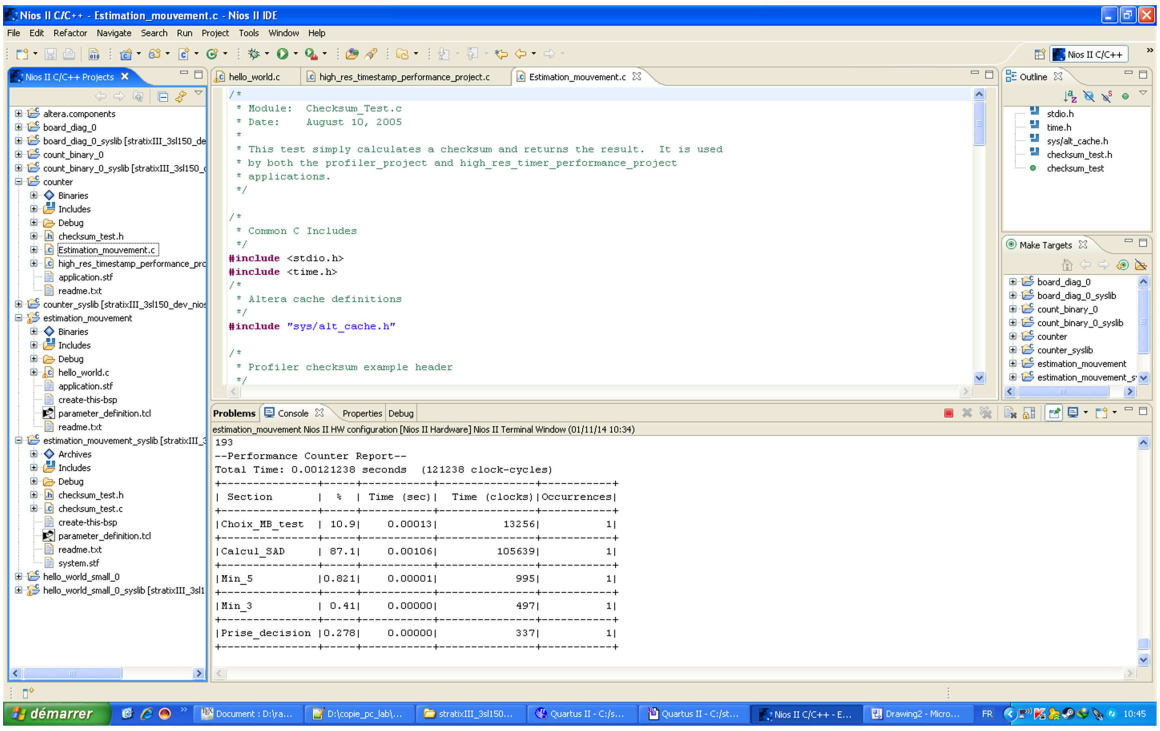

FIGURE 6.17 – Interface graphique de l'outil NIOS II IDE

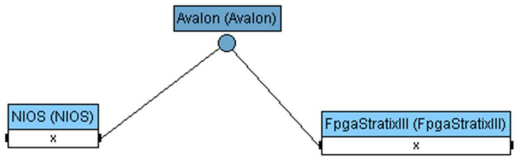

FIGURE 6.18 – Graphe d'architecture spécifié

```
--Performance Counter Report--
Total Time: 0.0026793 seconds  (267930 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|Choix_Position |  4.9|    0.00013|          13119|          1|
+---------------+-----+-----------+---------------+-----------+
|Calcul_SAD     | 39.6|    0.00106|         106107|          1|
+---------------+-----+-----------+---------------+-----------+
|Min_5          |0.296|    0.00001|            794|          1|
+---------------+-----+-----------+---------------+-----------+
|Min_3          |0.214|    0.00001|            573|          1|
+---------------+-----+-----------+---------------+-----------+
|Prise_decision |0.122|    0.00000|            326|          1|
+---------------+-----+-----------+---------------+-----------+
|Fenetre_Recherche| 42.6|    0.00114|         114066|          1|
+---------------+-----+-----------+---------------+-----------+
|MB_Courant     | 12.2|    0.00033|          32565|          1|
+---------------+-----+-----------+---------------+-----------+
```

FIGURE 6.19 – Résultats du profilage de l'estimateur de mouvement

Le composant "Performance counter" permet de donner le résultat du profilage en terme de temps de calcul, de nombre de cycles, le nombre d'appels à la fonction et le

pourcentage que prend la fonction par rapport au temps total d'exécution. Le code utilisé pour ce profilage contient une seule occurrence de chaque opération, donc les pourcentages données dans le rapport ne reflètent pas les pourcentages exacts du temps d'exécution par rapport à la latence totale de l'estimateur de mouvement. On remarque que les temps d'exécution des opérations "Min3" et "PriseDecision" indiqués dans le rapport de profilage sont nuls (trop faibles). Pour cette raison, et comme tout les éléments de l'architecture fonctionnent avec le même signal d'horloge, nous allons utiliser une spécification temporelle proportionnelle au nombres de cycles nécessaires pour exécuter les différentes opérations. On utilise 32 bits du bus Avalon pour le transfert de données entre le processeurs et le FPGA, donc on peut l'utiliser pour transférer 4 pixels (uchar) à la fois. C'est aussi pourquoi nous avons décidé de multiplier tout les nombres de cycles nécessaires pour les différentes opérations par 4. On obtient ainsi les spécifications temporelles présentées dans le tableau 6.1. On remarque que les performances de traitement du composant reconfigurable sont nettement meilleures que celles du processeur NIOS II, ce qui est prévisible.

Tableau 6.1 – Spécifications temporelles des opérations de l'estimateur de mouvement sur les différents opérateurs

Opération	NIOS II	FPGAStratixIII
FenetreRecherche	456264	
MBCourant	130260	
ChoixPosition	52476	12
CalculSAD	424428	2624
Min5	3176	64
PriseDecision	1304	12
Min3	2292	12
AbsoluteDifference		8
Accumulation		512
CalculPositionCentrale	929	12
PositionCentrale	1	1
VecteurMouvement	256	

6.5.3 Résultats obtenus

Puisque le FPGA peut exécuter la totalité des opérations du graphe d'algorithme (mise à part les entrées/sorties), le graphe d'architecture modifié comporte 21 opérateurs dégénérés pour le calcul. Le graphe d'architecture obtenu est représenté par la figure 6.20

Les résultats de partitionnement, obtenus à partir du fichier intermédiaire, sont donnés dans le tableau 6.2. On remarque que seules les opérations d'entrées/sorties sont distribuées sur le processeur NIOS II. En effet, toutes les opérations de traitement sont distribuées sur le composant reconfigurable. Cette distribution permet d'avoir le vecteur de mouvement au bout de 590528 unités de temps si la position centrale correspond au minimum de SAD dés la première itération. Si non, avec cette distribution chaque itération supplémentaire coute 5373 unités de temps. Donc le vecteur de mouvement est trouvé en (590528 + (N-1) x 5373) unités de temps avec N le nombre d'itérations nécessaires.

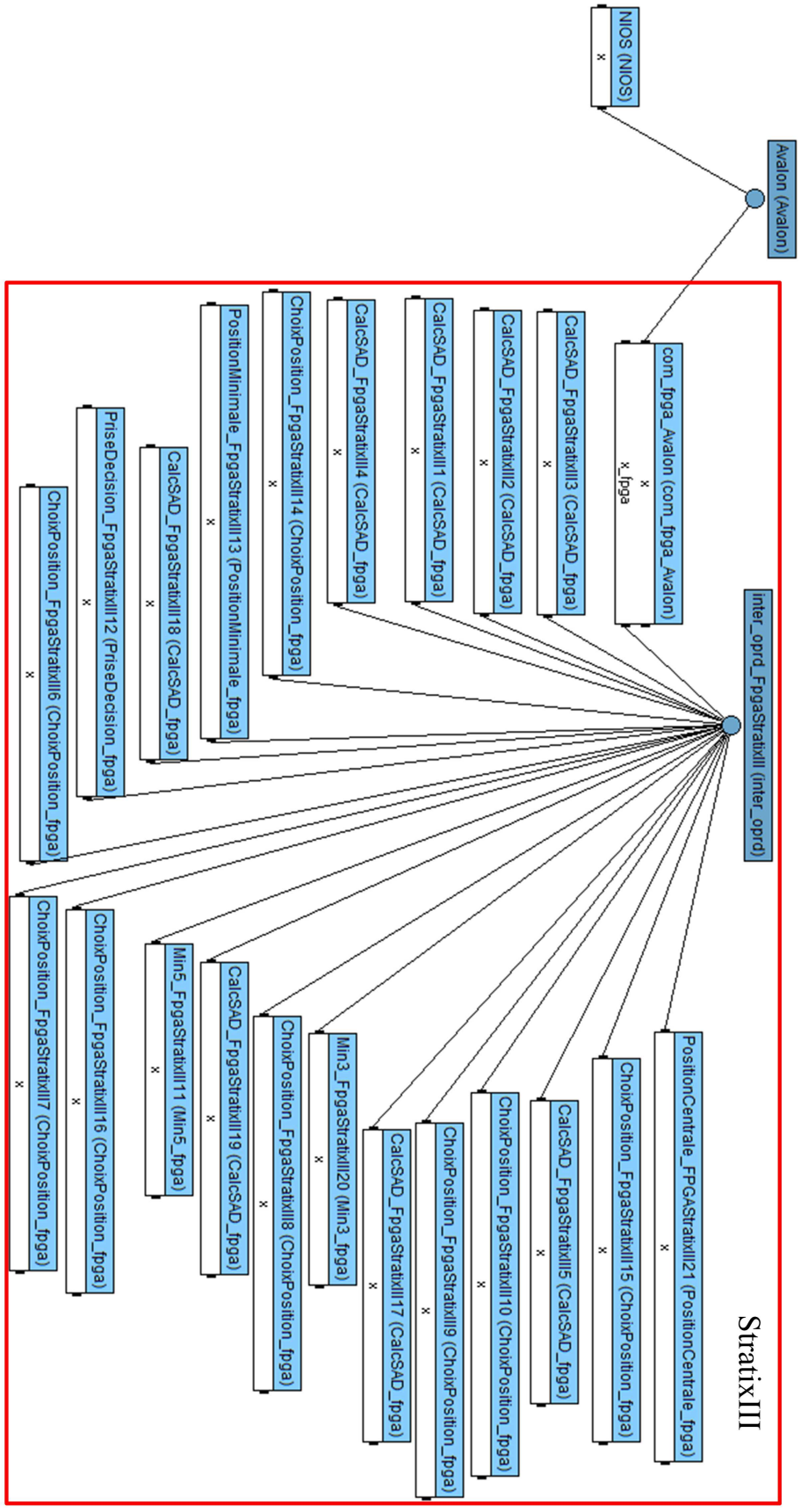

FIGURE 6.20 – Graphe d'architecture transformé

Tableau 6.2 – Résultats du partitionnement

Opération	Date de début	Date de fin	Opérateur
MacroblocCourant	0	130260	NIOS
FenetreRecherche	130260	586524	NIOS
ChoixMBTestCentral	587544	587556	FPGAStratixIII
ChoixMBTestGauche	587544	587556	FPGAStratixIII
ChoixMBTestDroite	587544	587556	FPGAStratixIII
ChoixMBTestBas	587544	587556	FPGAStratixIII
ChoixMBTestHaut	587544	587556	FPGAStratixIII
CalcSADc	587556	590180	FPGAStratixIII
CalcSADg	587556	590180	FPGAStratixIII
CalcSADd	587556	590180	FPGAStratixIII
CalcSADb	587556	590180	FPGAStratixIII
CalcSADh	587556	590180	FPGAStratixIII
Minimum	590180	590245	FPGAStratixIII
PriseDeDecision	590245	590256	FPGAStratixIII
ChoixMBTestD1	590256	590268	FPGAStratixIII
ChoixMBTestD2	590256	590268	FPGAStratixIII
ChoixMBTestD3	590256	590268	FPGAStratixIII
CalculSADD1	590268	592892	FPGAStratixIII
CalculSADD2	590268	592892	FPGAStratixIII
CalculSADD3	590268	592892	FPGAStratixIII
PositionMinimale	592892	592904	FPGAStratixIII
PositionCentrale	592904	592916	FPGAStratixIII
Centre	592916	592917	FPGAStratixIII
VecteurDeMouvement	590272	590528	NIOS

Malgré cette distribution, la solution proposée n'utilise que 15 des 21 opérateurs dégénérés créés dans le FPGA. En effet, comme le montre la figure 6.21, chaque paire des opérations "CalcSADd" et "CalcSADD3", "CalcSADg" et "CalcSADD2", "CalcSADb" et "CalcSADD1" est distribuée sur un seul opérateur dégénéré. Donc pour calculer les 8 valeurs de SAD de l'algorithme, on n'utilise que 5 opérateurs. De la même façon, pour effectuer les choix des 8 macroblocs tests, on n'utilise que 5 opérateurs dégénérés puisque chaque paire des opérations "ChoixMBTestCentral" et "ChoixMBTestD1", "ChoixMBTestHaut" et "ChoixMBTestD2", "ChoixMBTestBas" et "ChoixMBTestD3" est distribuée sur un seul opérateur dégénéré.

Après optimisation, 588432 unités de temps sont nécessaires si on obtient le vecteur de mouvement dès la première itération. Ce qui correspond à une défactorisation totale des frontières de factorisation contenues dans toutes les opérations SAD. Ce résultat représente une réduction de 2096 unités de temps, soit un gain d'environ 0,4%. Ce gain est faible à cause de la durée énorme de la lecture des données. Le nombre d'unités de temps nécessaire pour chaque itération supplémentaire passe de 5373 à 1181, ce qui correspond à un gain de 78%. Ainsi, après optimisation, le vecteur de mouvement est produit après (588432 + (N-1) x 1181) unités de temps avec N le nombre d'itérations nécessaires.

Le code VHDL généré par SynDEx-IC est intégré au système en utilisant SOPC Builder. La synthèse de ce système complet consomme 13153 unités logiques, ce qui représente 12% des 113600 unités logiques disponibles dans le FPGA.

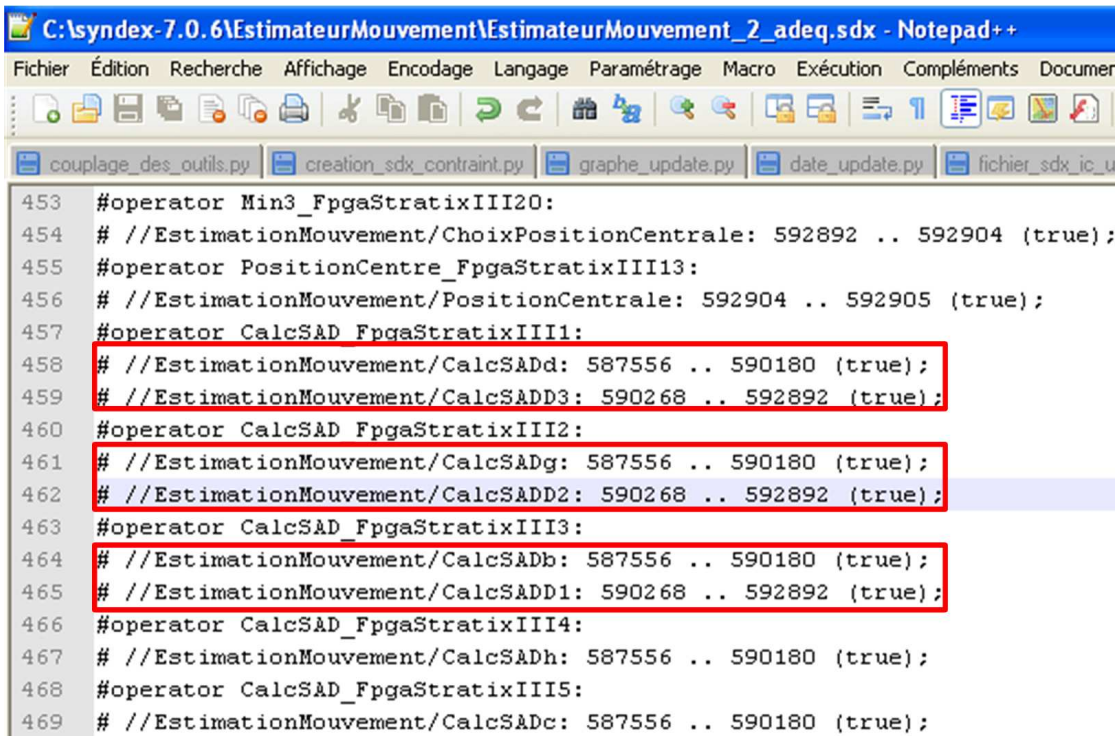

FIGURE 6.21 – Imprime écran du fichier de partitionnement

Cet exemple n'est pas si intéressant car la totalité de l'estimateur de mouvement tient dans le FPGA. Nous avons cependant pu vérifier la bonne collaboration entre SynDEx et SynDEx-IC et valider notre générateur automatique de code mixte.

6.5.4 Conclusion

Ce chapitre décrit les scriptes permettant d'automatiser l'algorithme de couplage des outils SynDEx et SynDEx-IC. En effet, cet algorithme manipule les fichiers sdx, nécessaires au lancement de SynDEx et SynDEx-IC, en extrait des données, les transforme pour créer de nouvelles données dans de nouveaux fichiers. Ces scriptes sont rédigés en Python pour une manipulation des fichiers et chaines de caractères plus facile. l'algorithme de couplage des outils

Nous avons présenté aussi l'estimateur de mouvement de la norme de codage vidéo H.264/AVC. Cette étape est cruciale dans la compression vidéo car elle permet une grande efficacité de compression. Mais une grande capacité de calcul est nécessaire pour aboutir à un grand taux de compression. Nous avons décidé d'implémenter cet estimateur de mouvement sur une architecture mixte composée d'un processeur NIOS II et d'un FPGA Stratix III. Nous avons utilisé notre outil basé sur la méthodologie AAA pour effectuer cette implantation et obtenir un système combinant les points forts du processeur et ceux des FPGA.

Chapitre 7

Conclusion générale

Les architectures mixtes comportant à la fois des composants programmables et d'autres reconfigurables représentent une solution qui permet d'apporter la puissance de traitement nécessaires à l'implémentation temps réel des applications évoluées d'aujourd'hui. Mais cette puissance est associée à une complexité de conception élevée. Notre but dans cette thèse est la mise au point d'une méthodologie et d'un outil d'aide à la conception de systèmes temps réel basés sur les architectures mixtes.

En premier lieu, nous avons présenté un état de l'art des architectures permettant l'implantation des applications temps réel. Nous nous sommes essentiellement intéressés aux différents types d'architectures matérielles de traitement de données. Nous avons alors présenté les architectures programmables, les architectures reconfigurables et les architectures mixtes. Nous avons aussi présenté un état de l'art des outils et méthodes de développement des systèmes temps réel. En effet, ces méthodes permettent de traiter une ou plusieurs étapes de la conception, à savoir la modélisation, la simulation, l'exploration de l'espace des solutions et la génération automatique de codes.

Nous avons présenté, ensuite, la méthodologie de prototypage rapide Adéquation Algorithme Architecture (AAA). Elle permet de diminuer considérablement le temps nécessaire pour la conception de systèmes embarqués temps réel basés sur les architectures multicomposants programmables. Cette méthodologie est implantée dans l'outil de conception assisté par ordinateur SynDEx qui permet d'avoir une implémentation optimisée d'un algorithme sur une architecture multicomposant programmable. L'algorithme est modélisé par un graphe factorisé et conditionné de dépendance de données (GFCDD). Dans ce graphe, chaque sommet représente une opération et chaque arc représente une dépendance de donnée ou de contrôle entre les opérations. L'architecture cible est représentée par un graphe où chaque sommet représente un composant de calcul (processeur, ASIC, ...) ou un composant de communication et chaque arc représente une connexion bidirectionnelle. L'adéquation est effectuée en utilisant une heuristique gloutonne qui réalise une analyse d'ordonnancement distribué et cherche à minimiser le temps d'éxécution de l'algorithme sur le multicomposant en tenant compte des coûts de communications interprocesseur. SynDEx permet non seulement de trouver en un temps relativement court une implémentation optimisée de l'algorithme sur l'architecture, mais aussi de générer automatiquement un exécutif générique pour chaque composant programmable du graphe d'architecture.

La méthodologie AAA a été étendue pour l'optimisation des algorithmes sur les composants reconfigurables (FPGA). Cet extension utilise la même représentation des algorithmes. Par contre, la spécification de l'architecture se réduit à la liste des opérations que le FPGA peut exécuter et pour chacune de ces opérations la durée d'exécution et

du nombre de blocs logiques utilisés. Dans cette extension, l'utilisateur spécifie aussi une contrainte temporelle que l'algorithme doit respecter. L'optimisation est réalisée par défactorisation partielle ou totale de l'algorithme (initialement factorisé). Cela correspond à la technique classique de déroulage de boucle sur processeur sachant que dans le cas des FPGA la défactorisation entraîne un gain en latence mais un coût en nombre de blocs logiques plus élevé. Une heuristique est utilisée pour choisir la boucle à dérouler et son degré de déroulage. L'implémentation obtenue satisfait alors la contrainte temporelle imposée (ou s'en approche au maximum s'il est impossible de la satisfaire) en utilisant le minimum de blocs logiques du FPGA. Cette extension est utilisée dans l'outil de conception assisté par ordinateur SynDEx-IC qui permet en plus de l'optimisation de l'implantation de générer automatiquement le code VHDL de cette implantation.

Malheureusement, ni la méthodologie AAA, ni son extension ne couvrent les architectures mixtes formées par des composants programmables et des composants reconfigurables. En effet, la méthodologie AAA cible les architectures multi-composants programmables et son extension cible les architectures reconfigurables (mono-FPGA). Nous avons donc proposé une extension du modèle d'architecture de AAA pour pouvoir modéliser les architectures mixtes en tenant compte des spécificités de chaque type de composant. Puis une extension du modèle d'implantation AAA est présentée pour pouvoir tenir compte du parallélisme massif offert par les composants reconfigurables.

Nous avons effectué l'étude des principaux algorithmes de partitionnement pour le co-design. Ces algorithmes sont comparés à l'algorithme de partitionnement que nous proposons. Notre algorithme permet de passer du modèle d'architecture AAA non étendu vers notre extension de ce modèle. Puis nous utilisons SynDEx pour le partitionnement des opérations sur les composants programmables et reconfigurables de l'architecture. Enfin SynDEx-IC est utilisé pour l'optimisation de l'implantation des parties distribuées sur les composants reconfigurables.

Dans le cadre de la génération automatique des codes pour les architectures mixtes, nous avons proposé une IP pour gérer la communication et la synchronisation des communications et des calculs dans les composants reconfigurables. Cette IP de communication est générique, elle est générée à partir des macro codes produits par SynDEx. Pour s'assurer que cette IP de communication fonctionne correctement et qu'elle n'occupe pas une grande surface du FPGA, nous l'avons synthétisé pour un exemple de liste de communications.

L'algorithme de partitionnement et d'optimisation des applications sur les architectures mixtes que nous avons proposé, est réalisé grâce à des scripts écrits en Python. L'ensemble de ces scripts est décrit dans ce rapport de thèse.

Enfin, nous avons validé ce travail en implantant un module de l'encodeur vidéo H.264. Cet encodeur a donc été présenté en détail en nous intéressant en particulier à l'estimation de mouvement. Nous avons présenté plusieurs recherches visant la réduction de la latence de cette étape importante de la norme H.264. Nous avons utilisé notre algorithme de partitionnement des applications sur les architectures mixtes pour optimiser l'implantation de l'estimateur de mouvement de H.264 sur une architecture mixte composée du processeur NIOS II et d'un FPGA stratix 3.

En conclusion, nous avons créé un outil de partitionnement et d'optimisation ciblant les architectures mixtes basées sur la méthodologie Adéquation Algorithme Architecture. Cet outil utilise le modèle d'architecture de AAA que nous avons étendu pour pouvoir tenir compte des différences entre les composants programmables et les composants reconfigurables. Il fait appel à SynDEx pour le partitionnement et l'ordonnancement des

opérations sur les composants de l'architecture et à SynDEx-IC pour l'optimisation de l'implantation des parties distribuées sur les composants reconfigurables. Notre outil permet aussi de générer automatiquement les codes séquentiels pour les processeurs, les codes VHDL pour le(s) FPGA et les communications entre les différents composants de l'architecture.

A la suite de ces travaux, nous envisageons une amélioration des différentes étapes de l'algorithme de couplage des outils SynDEx et SynDEx-IC. Nous pouvons améliorer la génération des codes VHDL en tenant compte du partitionnement donné par SynDEx. Les étapes de partitionnement et d'optimisation peuvent éventuellement être améliorées en faisant des appels itératifs à SynDEx et SynDEx-IC mais une étude approfondie doit être faite pour ne pas tomber sur un minimum local et se bloquer dans une boucle infinie. Il sera aussi possible d'envisager l'utilisation d'heuristique basée sur du recuit simulé, mais il faudra alors prendre garde aux temps très long que prendra alors l'heuristique, temps incompatible avec le prototypage rapide que nous nous sommes fixés.

Bibliographie

[Actel,] Actel. http ://www.actel.com/.

[Altera, a] Altera. http ://www.altera.com/.

[Altera, b] Altera. http ://www.altera.com/products/selector/psg-selector.html#.

[Altera, 2006] Altera (2006). White Paper FPGA Architecture. Technical Report July.

[Altera, 2011] Altera (2011). Profiling Nios II Systems. Technical Report July.

[Altera, 2013a] Altera (2013a). Altera's user-customizable ARM-based SoC. Technical report.

[Altera, 2013b] Altera (2013b). Stratix III 3SL150 Development Board. Technical report.

[Altera, 2014a] Altera (2014a). Avalon Interface Specifications. Technical report.

[Altera, 2014b] Altera (2014b). Cyclone V SoC Development Board. Technical report.

[ARM, 2011] ARM (2011). RealView LT-XC5VLX330. Technical report.

[Asthana, 2010] Asthana, R. M. (2010). *High-Level CSP Model Compiler for FPGAs.* PhD thesis.

[Ben Ayed et al., 2007] Ben Ayed, M. A., Samet, A., and Masmoudi, N. (2007). Toward An Optimal Block Motion Estimation Algorithm For H.264/AVC. *International Journal of Image and Graphics*, 07(02) :303–320.

[Betker et al., 1997] Betker, M. R., Fernando, J. S., and Whalen, S. P. (1997). The History of the Microprocessor. *Bell Labs Technical Journal*, (1947) :29–56.

[Bjerregaard and Mahadevan, 2006] Bjerregaard, T. and Mahadevan, S. (2006). A Survey of Research and Practices of Network-on-Chip. *ACM Computing Survery*, 38(March 2006).

[Bossuet, 2004] Bossuet, L. (2004). *Exploration de l'espace de conception des architectures reconfigurables.* PhD thesis, Université de Bretagne Sud, Lorient.

[Brooks et al., 2007] Brooks, C., Lee, E. A., Liu, X., Neuendorffer, S., Zhao, Y., and Zheng, H. (2007). Heterogeneous Concurrent Modeling and Design in Java (Volume 1 : Introduction to Ptolemy II). Technical report.

[Ce Zhu et al., 2002] Ce Zhu, Xiao Lin, and Chau, L.-P. (2002). Hexagon-based search pattern for fast block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(5) :349–355.

[Chaouch et al., 2007] Chaouch, H., Werda, I., Samet, A., and Masmoudi, N. (2007). Search window impact on H.264/AVC motion search implementation on TMS320C6416 DSP. In *International Multi-Conference on Systems, Signals & Devices SSD.*

[Chiang et al., 1992] Chiang, S., Forouhi, R., Chen, W., Hawley, F., McCollum, J., Hamdy, E., and Hu, C. (1992). Antifuse structure comparison for field programmable gate arrays. In *international electron devices meeting IEDM*, pages 611–614.

[Chun-Ling Yang et al., 2004] Chun-Ling Yang, Lai-Man Po, and Wing-Hong Lam (2004). A fast H.264 intra prediction algorithm using macroblock properties. In *2004 International Conference on Image Processing, 2004. ICIP '04.*, volume 1, pages 461–464. IEEE.

[Corporation, 2009] Corporation, A. (2009). The ARM Cortex-A9 Processors. Technical report.

[Corporation, 2012] Corporation, I. (2012). Behavioral , Real-Time and Performance Analysis of Multicore Embedded Systems. Technical report.

[Damak et al., 2010] Damak, T., Werda, I., Ben Ayad, M. A., and Masmoudi, N. (2010). An efficient zero length prefix algorithm for H.264 CAVLC decoder on TMS320C64. In *5th International Conference on Design & Technology of Integrated Systems in Nanoscale Era*, pages 1–4. IEEE.

[Damak et al., 2011] Damak, T., Werda, I., Masmoudi, N., and Bilavarn, S. (2011). Fast prototyping H.264 Deblocking filter using ESL tools. In *Eighth International Multi-Conference on Systems, Signals & Devices*, pages 1–4. IEEE.

[Dou et al., 2008] Dou, Y., Deng, L., Xu, J., and Zheng, Y. (2008). DMA Performance Analysis and Multi-core Memory Optimization for SWIM Benchmark on the Cell Processor. *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 170–179.

[Editor, 2014] Editor, C. P. (Ptolemy.org,2014). *System Design, Modeling, and Simulation Using Ptolemy II.*

[Eles et al., 1994a] Eles, P., Kuchcinski, K., Peng, Z., and Minea, M. (1994a). Synthesis of VHDL Concurrent Processes. In *Proceeding of the European Design Automation Conference (EURO-DAC), Grenoble, France.*

[Eles et al., 1994b] Eles, P., Peng, Z., and Doboli, A. (1994b). VHDL system-level specification and partitioning in a hardware/software co-synthesis environment. In *Third International Workshop on Hardware/Software Codesign*, pages 49–55. IEEE Comput. Soc. Press.

[Eles et al., 1996] Eles, P., Peng, Z., Kuchcinski, K., and Doboli, A. (1996). System Level Hardware / Software Partitioning Based on Simulated Annealing and Tabu Search. *Design Automation for Embedded Systems*, 2 :5–32.

[Farooq et al., 2012] Farooq, U., Marrakchi, Z., and Mehrez, H. (2012). *Tree-based Heterogeneous FPGA Architectures.* Springer New York, New York, NY.

[Feki et al., 2014a] Feki, O., Grandpierre, T., Akil, M., and Masmoudi, N. (2014a). Automatic Hardware/Software Interface Generation For SynDEx-mixte. In *2014 1st International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*, pages 512–516. IEEE.

[Feki et al., 2014b] Feki, O., Grandpierre, T., Akil, M., Masmoudi, N., and Sorel, Y. (2014b). SynDEx-Mix : A Hardware/Software Partitioning CAD Tool. In *15th international conference on Sciences and techniques of Automatic control & computer engineering*, pages 247–252.

[Feki et al., 2013] Feki, O., Grandpierre, T., Masmoudi, N., Akil, M., and Sorel, Y. (2013). Optimization of Real Time Application on Mixed Architecture Using AAA Methodology Extension. *International Journal of Electronics Communication and Computer Engineering*, 4(5) :1455–1466.

[Freund et al., 1997] Freund, L., Dupont, D., Israel, M., and Rousseau, F. (1997). Overview of the ECOS project. *Proceedings 8th IEEE International Workshop on Rapid System Prototyping Shortening the Path from Specification to Prototype.*

[Gallant and Kossentini, 1998] Gallant, M. and Kossentini, F. (1998). An efficient computation-constrained block-based motion estimation algorithm for low bit rate video coding. In *Conference Record of Thirty-Second Asilomar Conference on Signals, Systems and Computers (Cat. No.98CH36284)*, volume 1, pages 467–471. IEEE.

[Gamal et al., 1989] Gamal, A. E. L., Greene, J., Reyneri, J., Rogoyski, E., El Ayat, K., and Mohsen, A. (1989). An Architecture for Electrically Configurable Gate Arrays. *IEEE Journal of Solid-State Circuits*, 24(2) :394–398.

[Gouyet and Sablier, 2007] Gouyet, J.-N. and Sablier, M. (2007). MPEG-4 : Advanced Video Coding. Systèmes et applications. *Techniques de l'ingénieur.*

[Grandpierre, 2000] Grandpierre, T. (2000). *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.* PhD thesis, Paris XI.

[Grandpierre and Sorel, 2003] Grandpierre, T. and Sorel, Y. (2003). From algorithm and architecture specifications to automatic generation of distributed real-time executives : a seamless flow of graphs transformations. In *First ACM and IEEE International Conference on Formal Methods and Models for CoDesign MEMOCODE 03 Proceedings.*

[Guterman et al., 1979] Guterman, D. C., Rimawi, I. H., Chiu, T.-L., Halvorson, R. D., and Mcelroy, D. J. (1979). An electrically alterable nonvolatile memory structure using a floating-gate Cell. *IEEE Journal of Solid-State Circuits*, (2) :498–508.

[Han et al., 2013] Han, H., Liu, W., Jigang, W., and Jiang, G. (2013). Efficient Algorithm for Hardware / Software Partitioning and Scheduling on MPSoC. *Journal of computers*, 8(1) :61–68.

[Hsieh et al., 1988] Hsieh, H.-c., Dong, K., Ja, J. Y., Kanazawa, R., Ngo, L. T., Tinkey, L. G., Carter, W. S., and Freeman, R. H. (1988). A 9000-gate user-programmable gate array. In *Custom integrated circuits conference 1988*, pages 1–7.

[Instruments, 2012] Instruments, T. (2012). TMS320C6670 Multicore Fixed and Floating-Point System-on-Chip : data Manual. Technical Report March.

[ITU-T, 2003] ITU-T (2003). Advanced video coding for generic audiovisual services : ITU-T Recommendation H.264. Technical report.

[Kalla et al., 2010] Kalla, R., Sinharoy, B., Starke, W. J., and Floyed, M. (2010). Power 7 : IBM's next generation server processor. *Micro, IEEE*, 30(2) :7–15.

[Kaouane, 2004] Kaouane, L. (2004). *Formalisation et optimisation d'applications s'exécutant sur architecture reconfigurable.* PhD thesis, UPE-MLV.

[Kaouane et al., 2004] Kaouane, L., Akil, M., Grandpierre, T., and Sorel, Y. (2004). A methodology to implement real-time applications onto reconfigurable circuits. *Journal of Supercomputing*, 30(3) :362–376.

[Katz, 1994] Katz, R. H. (1994). *Contemporary logic design.* Benjamin/Cummings.

[Kessentini et al., 2008] Kessentini, A., Kaaniche, B., Werda, I., Samet, A., and Masmoudi, N. (2008). Complexity intra 16x16 prediction for H.264/AVC. In *International Conference on Embedded Systems & Critical Applications.*

[Kuon et al., 2008] Kuon, I., Tessier, R., and Rose, J. (2008). *FPGA Architecture : Survey and Challenges.* Now Publishers Inc.

[Lavarenne et al., 1991] Lavarenne, C., Seghrouchni, O., Sorel, Y., and Sorine, M. (1991). The SynDEx Software Environment for Real-Time Distributed Systems, Design and Implementation. In *Proceedings of European Control Conference, ECC'91*.

[Li et al., 2013] Li, S., Farahini, N., Hemani, A., Rosvall, K., and Sander, I. (2013). System Level Synthesis of Hardware for DSP Applications Using Pre-Characterized Function Implementations. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS),*, pages 1–10.

[Liou, 1994] Liou, M. (1994). A new three-step search algorithm for block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 4(4) :438–442.

[List et al., 2003] List, P., Joch, A., Lainema, J., Bjontegaard, G., and Karczewicz, M. (2003). Adaptive deblocking filter. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7) :614–619.

[Loukil et al., 2009] Loukil, H., Arous, S., Werda, I., Atitallah, A. B., Kadionik, P., and Masmoudi, N. (2009). Hardware architecture for H.264/AVC intra 16x16 frame processing. In *2009 6th International Multi-Conference on Systems, Signals and Devices*, pages 1–5. IEEE.

[Macii and Poncino, 1994] Macii, E. and Poncino, M. (1994). FPGA Synthesis Using Look-Up Table and Multiplexor Based Architectures. In *Mediterranean Electrotechnical Conference. MELECON*, pages 302–305.

[MathWorks, 2012] MathWorks (2012). Simulation and Model-Based Design, accessed on www.mathworks.com/products/simulink. Technical report.

[Mcclanahan, 2010] Mcclanahan, C. (2010). History and Evolution of GPU Architecture. pages 1–7.

[Moore, 1965] Moore, G. (1965). Cramming more components onto integrated circuits. 38(8).

[Niang et al., 2004] Niang, P., Grandpierre, T., Akil, M., and Sorel, Y. (2004). AAA and SynDEx-Ic : A Methodology and a Software Framework for the Implementation of Real-Time Applications onto Reconfigurable Circuits. *Field Programmable Logic and Application*, 3203 :1119–1123.

[NVIDIA, 2012] NVIDIA (2012). NVIDIA's Next Generation CUDA Compute Architecture : Kepler GK110.

[Richardson, 2002] Richardson, I. E. (2002). *Video Codec Design : Developing Image and Video Compression Systems.* John Wiley & Sons.

[Richardson, 2003] Richardson, I. E. (2003). *H.264 and MPEG-4 Video Compression : Video Coding for Next-generation Multimedia.* John wiley edition.

[Rousseau et al., 1995] Rousseau, F., Benzakki, J., Berge, J.-m., Israel, M., Coquibus, B., France, E. C., and Cnet, F. T. (1995). Adaptation of Force-Directed Scheduling Algorithm for Hardwarer ' Software Partitioning. In *Sixth IEEE International Workshop on Rapid System Prototyping, 1995. Proceedings.,*, pages 33–37.

[Sorel, 1994] Sorel, Y. (1994). Massively parallel computing systems with real time constraints, the algorithm architecture adequation methodology. In *the Massively Parallel Computing Systems.*

[Sorel, 1996] Sorel, Y. (1996). Real-Time Embedded Image Processing Applications using the algorithm architecture adequation Methodology. In *Proceedings of IEEE International Conference on Image Processing, ICIP'96*, Lausanne, Switzerland.

[Srinivasan et al., 1998] Srinivasan, V., Radhakrishnan, S., and Vemuri, R. (1998). Hardware software partitioning with integrated hardware design space exploration. *Proceedings Design Automation and Test in Europe*, pages 28–35.

[Staunstrup and Wolf, 1997] Staunstrup, J. and Wolf, W. (1997). *Hardware/Software Co-Design*. Springer Science & Business Media.

[Thiele et al., 2007] Thiele, L., Bacivarov, I., Haid, W., and Huang, K. (2007). Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*, pages 29–40. IEEE.

[Tong et al., 2006] Tong, J. G., Anderson, I. D. L., and Khalid, M. a. S. (2006). Soft-Core Processors for Embedded Systems. In *18th International Conference on Microelectronics*, number Icm, pages 170–173. Ieee.

[Tsay, 2000] Tsay, J. (2000). A Code Generation Framework for Ptolemy II.

[Vicard and Sorel, 1998] Vicard, A. and Sorel, Y. (1998). Formalization and static optimization for parallel implementations. In *Proceedings of Workshop on Distributed and Parallel Systems, DAPSYS'98*, Budapest, Hungary.

[Werda, 2011] Werda, I. (2011). *Implémentation et optimisation de l'encodeur Baseline H.264/AVC sur la plateforme TI C64x*. PhD thesis, Sfax University, Tunisia.

[Werda et al., 2007] Werda, I., Chaouch, H., Samet, A., Ben Ayad, M. A., and Masmoudi, N. (2007). Optimal DSP-Based Motion Estimation Tools Implementation For H . 264 / AVC Baseline Encoder. *IJCSNS International Journal of Computer Science and Network Security*, 7(5) :141–150.

[Werda et al., 2010] Werda, I., Chaouch, H., Samet, A., Ben Ayed, M., and Masmoudi, N. (2010). Optimal DSP Based Integer Motion Estimation Implementation for H . 264 / AVC Baseline Encoder. *The International Arab Journal of Information Technology*, 7(1) :96–104.

[Xilinx, a] Xilinx. http ://www.xilinx.com/.

[Xilinx, b] Xilinx. http ://www.xilinx.com/products/silicon-devices/fpga/index.htm.

[Xilinx, 2007] Xilinx (2007). Advantages of the Virtex-5 FPGA 6-Input LUT Architecture. Technical report.

[Xilinx, 2008] Xilinx (2008). Virtex-4 FX12 PowerPC and MicroBlaze Edition Kit Reference Systems. Technical report.

[Xilinx, 2009a] Xilinx (2009a). Summary of Virtex-5 FPGA Features. Technical report.

[Xilinx, 2009b] Xilinx (2009b). Virtex-5 FXT PowerPC and MicroBlaze Kit Reference Systems. Technical report.

[Xilinx, 2013a] Xilinx (2013a). Introduction to FPGA Design with Vivado High-Level Synthesis. Technical report.

[Xilinx, 2013b] Xilinx (2013b). Zynq-7000 All Programmable SoC Overview. Technical report.

[Zhao et al., 2013] Zhao, X., Zhang, H., Jiang, Y., Song, S., Jiao, X., and Gu, M. (2013). An Effective Heuristic-Based Approach for Partitioning. *Journal of Applied Mathematics*, 2013 :1–8.

[Zhu and Ma, 2000] Zhu, S. and Ma, K. K. (2000). A new diamond search algorithm for fast block-matching motion estimation. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 9(2) :287–90.

Annexe A

Code VHDL de l'IP de communication

Compteur

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.ALL;

ENTITY compteur1 IS

generic (
initialisation: std_logic;
modulo: integer;
output_width: integer
);
PORT
(

q : out STD_LOGIC_vector (output_width-1 downto 0);
init, clk : in STD_LOGIC
);
END compteur1;


ARCHITECTURE SYN OF compteur1 IS
signal int: unsigned (output_width-1 downto 0);

BEGIN
process (init, clk)
begin
if init = '1' then
int<= (others=>initialisation);
elsif (rising_edge(clk)) then
if (int = modulo-1) then
```

```
int<= (others=>initialisation);
else
int<= int + 1 ;
end if;
end if;

end process;
q<=std_logic_vector (int);
END SYN;
```

Comparateur

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.ALL;


ENTITY comparateur IS

 generic
(
data_width : natural
);
PORT
(
clk, init : in std_logic;
q : out STD_LOGIC;
data1, data2 : in STD_LOGIC_vector (data_width-1 downto 0)
);
END comparateur;


ARCHITECTURE SYN OF comparateur IS


BEGIN
process (data1, data2,clk,init)
begin
if init='1' then
q<='0';
elsif rising_edge(clk) then
if data2 = data1 then
q<='1';
else
q<='0';
```

```
end if;
end if;
end process;


END SYN;
```

ROM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.ALL;

ENTITY rom1 IS
generic (
mem_size, mem_width, adr_width: integer
);
PORT
(
q : out STD_LOGIC_vector (mem_width-1 downto 0);
adr : in STD_LOGIC_vector (adr_width-1 downto 0)
);
END rom1;


ARCHITECTURE SYN OF rom1 IS

type mem is array ( 0 to mem_size-1) of std_logic_vector (mem_width-1 downto 0);

-- Defining mem values
constant memoire: mem:=(
"00000010000",
"10000010000",
"11000000001",
"10100000000",
"11000000100");
BEGIN

-- Assigning memory output
q<=memoire(conv_integer (adr));

END SYN;
```

Machine à états finis

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;



ENTITY fsm IS
PORT
(

reset,start, ack_v_cpu,  req_v_cpu, wr: out STD_LOGIC;
clk, req_de_oprd, ack_de_cpu, req_de_cpu, fin ,init,in_out : in STD_LOGIC
);
END fsm;


ARCHITECTURE SYN OF fsm IS

type etat is (s0, s1, s2, s3, s4, s5);
signal etat_actuel: etat;


BEGIN

p1:process (init, clk)
begin
if init='1' then etat_actuel <= s0;
elsif rising_edge (clk) then
case etat_actuel is
when s0=>
wr<='0';
start<='0';
ack_v_cpu<='0';
req_v_cpu<='0';
      reset<='0';
if (req_de_cpu='1' and in_out='0') then
etat_actuel<=s1;

elsif (req_de_oprd='1' and in_out='1') then
etat_actuel<=s4;

      else
etat_actuel<=s0;
end if;
when s1=>
```

```
wr<='1';
start<='1';
ack_v_cpu<='1';
if req_de_cpu='0' then
etat_actuel<=s2;
wr<='0';
else
etat_actuel<=s1;
end if;
when s2=>
start<='0';
ack_v_cpu<='0';
wr<='0';
if  fin='1' then
etat_actuel<=s3;

elsif (fin='0' and req_de_cpu='1' and in_out='0') then
etat_actuel<=s1;

      else
etat_actuel<=s2;
end if;
when s3=>
wr<='0';
      reset<='1';
      start<='0';
etat_actuel<=s0;

when s4=>
req_v_cpu<='1';
        start<='1';
if ack_de_cpu='1' then
etat_actuel<=s5;

else
etat_actuel<=s4;
end if;
when s5=>
req_v_cpu<='0';
start<='0';
if fin='1' then
etat_actuel<=s3;

      elsif (req_de_oprd='1' and in_out='1') then
      etat_actuel<=s4;


else
```

```
etat_actuel<=s5;
end if;
   when others=>
   etat_actuel<=s0;
end case;
end if;

end process;


END SYN;
```

Mux_in_out

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY mux_in_out IS
generic
(
bus_widht: integer
);
PORT
(
sel : in std_logic;
data_bus_com : inout STD_LOGIC_vector (bus_widht-1 downto 0);
data_vers_oprd : out STD_LOGIC_vector (bus_widht-1 downto 0);
data_de_oprd : in STD_LOGIC_vector (bus_widht-1 downto 0)
);
END mux_in_out;


ARCHITECTURE SYN OF mux_in_out IS


BEGIN
process (sel, data_bus_com, data_de_oprd)
begin

data_vers_oprd <= data_bus_com;

if (sel = '0') then
-- High impedecy to manage inout port
data_bus_com  <= (others=>'Z');
```

```
else
data_bus_com <= data_de_oprd;
end if;

end process;


END SYN;
```

Oprd_synch

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;



ENTITY oprd_synch IS
generic (
nbr_req_sen: integer;
req_sen_width: integer;
nbr_ack_sen: integer;
ack_sen_width: integer
);
PORT
(
ack_t_oprd, req_t_oprd: out STD_LOGIC;
synchro, in_out, en, req_f_oprd, ack_f_oprd, init, clk: in STD_LOGIC;
req_demux_cmd: out std_logic_vector (req_sen_width-1 downto 0);
ack_demux_cmd: out std_logic_vector (ack_sen_width-1 downto 0)
);
END oprd_synch;


ARCHITECTURE SYN OF oprd_synch IS

component compteur1 IS

generic (
initialisation: std_logic;
modulo: integer;
output_width: integer
```

```
);
PORT
(

q : out STD_LOGIC_vector (output_width-1 downto 0);
init, clk : in STD_LOGIC
);
END component;

signal ack, req: std_logic;

BEGIN
process (clk, init)
begin
if init='1' then
ack<='0';
req<='0';
ack_t_oprd<='0';
req_t_oprd<='0';
elsif rising_edge(clk) then
if en='1' then
ack_t_oprd<=ack;
req_t_oprd<=req;
elsif (synchro='1' and in_out='0') then
req<='1';
elsif (synchro='1' and in_out='1') then
ack<='1';
end if;

if ack_f_oprd='1' then
req<='0';
req_t_oprd<='0';
end if;

if req_f_oprd='0' then
ack<='0';
ack_t_oprd<='0';
end if;

end if;

end process;

compt_sen_req: compteur1 generic map ('1',nbr_req_sen,req_sen_width) port map (req_de
compt_sen_ack: compteur1 generic map ('1',nbr_ack_sen,ack_sen_width) port map (ack_de

end syn;
```

Demux

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.ALL;


ENTITY demux IS
generic (
output_width: integer ;
sel_width: integer
);
PORT
(

input : in STD_LOGIC;
sel : in STD_LOGIC_vector (sel_width-1 downto 0);
output : out STD_LOGIC_vector (output_width-1 downto 0)

);
END demux;


ARCHITECTURE SYN OF demux IS


BEGIN
process (sel, input)
begin
for i in 0 to output_width-1 loop
if i=(conv_integer (unsigned(sel))) then
output (i)<= input;
else
output (i)<= '0';
end if;
end loop;
end process;


END SYN;
```

Rec_reg

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;



ENTITY rec_reg IS
generic
(
reg_width: integer
);
PORT
(
clk,wr : in std_logic;
data_out : out STD_LOGIC_VECTOR (reg_width-1 downto 0);
data_in : in STD_LOGIC_VECTOR (reg_width-1 downto 0)
);
END rec_reg;


ARCHITECTURE SYN OF rec_reg IS

signal valeur : STD_LOGIC_VECTOR (reg_width-1 downto 0):= (others=>'0');

BEGIN

data_out<= valeur;

process (wr,clk)
begin
if (rising_edge(clk)) then
if wr='1' then
valeur<=data_in;
end if;
end if;
end process;
END SYN;
```

Mux_n_input

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
```

```
USE ieee.std_logic_unsigned.ALL;



ENTITY mux_n_input IS
generic (
input_nbr: integer;
sel_width: integer
);
PORT
(

input : in STD_LOGIC_vector (input_nbr-1 downto 0);
sel : in STD_LOGIC_vector (sel_width-1 downto 0);
output : out STD_LOGIC

);
END mux_n_input;


ARCHITECTURE SYN OF mux_n_input IS

signal indice,indice1 : integer;
BEGIN

process (sel,input)
begin
if conv_integer(sel)<0 or conv_integer(sel)>input_nbr-1 then
output<= '0';
else
output <= input (conv_integer(sel));
end if;
end process;


END SYN;
```

Mux_generic

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.ALL;
```

```
ENTITY mux_generic IS
generic (
input_nbr: integer ;
output_width: integer ;
sel_width: integer
);
PORT
(

input : in STD_LOGIC_vector (input_nbr*output_width-1 downto 0);
sel : in STD_LOGIC_vector (sel_width-1 downto 0);
output : out STD_LOGIC_vector (output_width-1 downto 0)

);
END mux_generic;


ARCHITECTURE SYN OF mux_generic IS

component mux_n_input IS
generic (
input_nbr: integer ;
sel_width: integer
);
PORT
(

input : in STD_LOGIC_vector (input_nbr-1 downto 0);
sel : in STD_LOGIC_vector (sel_width-1 downto 0);
output : out STD_LOGIC

);
END component;

signal input1: STD_LOGIC_vector (input_nbr*output_width-1 downto 0);

BEGIN

l0: for j in 0 to output_width-1 generate
l1: for i in 0 to input_nbr-1 generate
input1(j*input_nbr+i)<= input(j+(i*output_width));
end generate l1;
end generate l0;

l2: for i in 0 to output_width-1 generate
muxi: mux_n_input generic map (input_nbr,sel_width) port map (input1((i+1)*input_nbr-1
end generate l2;
```

```
END SYN;
```

IP de communication

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY com IS
generic (
max_pck_width: integer :=9;
max_pck_nbr: integer:=256;
data_nbr: integer :=5;
rec_pck_nbr_width: integer :=9;
sen_pck_nbr_width: integer :=3;
data_nbr_width: integer :=3;
rec_pck_nbr: integer :=288;
sen_pck_nbr: integer :=5;
com_bus_width: integer :=8;
data_in_width: integer :=40;
data_out_width: integer :=2304;
nbr_req_sen: integer:=2;
req_sen_width: integer:=2;
nbr_ack_sen: integer:=2;
ack_sen_width: integer:=2
);
PORT
(  data_in : in std_logic_vector (data_in_width-1 downto 0);
data_out: out std_logic_vector (data_out_width-1 downto 0);
com_bus: inout std_logic_vector (com_bus_width-1 downto 0);
ack_v_oprd: out std_logic_vector (nbr_ack_sen-1 downto 0);
ack_v_cp, req_v_cp: out STD_LOGIC;
req_v_oprd: out std_logic_vector (nbr_req_sen-1 downto 0);
clk,req_d_oprd, ack_d_cpu, req_d_cpu, ack_d_oprd ,init : in STD_LOGIC
);
END com;


ARCHITECTURE SYN OF com IS


component comparateur IS
 generic
```

```
(
data_width : natural
);
PORT
(
clk, init : in std_logic;
q : out STD_LOGIC;
data1, data2 : in STD_LOGIC_vector (data_width-1 downto 0)
);
END component;

component compteur1
generic (
initialisation: std_logic;
modulo: integer;
output_width: integer
);
PORT
(

q : out STD_LOGIC_vector (output_width-1 downto 0);
init, clk : in STD_LOGIC
);
END component;




component rom1

generic (
mem_size, mem_width, addr_width: integer
);
PORT
(

q : out STD_LOGIC_vector (mem_width-1 downto 0);
adr : in STD_LOGIC_vector (addr_width-1 downto 0)
);
END component;

component fsm
PORT
(

reset,start, ack_v_cpu,  req_v_cpu, wr: out STD_LOGIC;
clk, req_de_oprd, ack_de_cpu, req_de_cpu, fin ,init,in_out : in STD_LOGIC
);
```

```
END component;

component mux_in_out
generic
(
bus_widht: integer
);
PORT
(
sel : in std_logic;
data_bus_com : inout STD_LOGIC_vector (bus_widht-1 downto 0);
data_vers_oprd : out STD_LOGIC_vector (bus_widht-1 downto 0);
data_de_oprd : in STD_LOGIC_vector (bus_widht-1 downto 0)
);
END component;

component oprd_synch IS

generic (
nbr_req_sen: integer;
req_sen_width: integer;
nbr_ack_sen: integer;
ack_sen_width: integer
);
PORT
(
ack_t_oprd, req_t_oprd: out STD_LOGIC;
synchro, in_out, en, req_f_oprd, ack_f_oprd, init, clk: in STD_LOGIC;
req_demux_cmd: out std_logic_vector (req_sen_width-1 downto 0);
ack_demux_cmd: out std_logic_vector (ack_sen_width-1 downto 0)
);

END component;

component demux IS
generic (
output_width: integer;
sel_width: integer
);
PORT
(

input : in STD_LOGIC;
sel : in STD_LOGIC_vector (sel_width-1 downto 0);
output : out STD_LOGIC_vector (output_width-1 downto 0)

);
END component;
```

```
component rec_reg IS
generic
(
reg_width: integer
);
PORT
(
clk,wr : in std_logic;
data_out : out STD_LOGIC_VECTOR (reg_width-1 downto 0);
data_in : in STD_LOGIC_VECTOR (reg_width-1 downto 0)
);
END component;

component mux_generic IS
generic (
input_nbr: integer ;
output_width: integer ;
sel_width: integer
);
PORT
(

input : in STD_LOGIC_vector (input_nbr*output_width-1 downto 0);
sel : in STD_LOGIC_vector (sel_width-1 downto 0);
output : out STD_LOGIC_vector (output_width-1 downto 0)

);
END component;

component and_gate IS

PORT
(
q : out STD_LOGIC;
data1, data2 : in STD_LOGIC
);
END component;

signal v1: std_logic_vector (max_pck_width-1 downto 0);
signal ve: std_logic_vector (max_pck_width+1 downto 0);
signal not_ve_max_pck_width,res ,egal ,ini, st, fini, wr, rd, compt_receiv_increment,
 compt_send_increment, ack_v_opr, req_v_opr : std_logic;
signal v2: std_logic_vector (data_nbr_width-1 downto 0);
signal rec_pck: std_logic_vector (rec_pck_nbr_width-1 downto 0);
signal sen_pck: std_logic_vector (sen_pck_nbr_width-1 downto 0);
signal wr_cmd: std_logic_vector (rec_pck_nbr-1 downto 0);
signal rd_cmd: std_logic_vector (sen_pck_nbr-1 downto 0);
```

```
signal sign_to_oprd, sign_from_oprd: std_logic_vector (com_bus_width-1 downto 0);
signal req_demux_cmd: std_logic_vector (req_sen_width-1 downto 0);
signal ack_demux_cmd: std_logic_vector (ack_sen_width-1 downto 0);


BEGIN
ini <= res or init;
not_ve_max_pck_width <= not(ve(max_pck_width));
and_rec: and_gate port map (compt_receiv_increment,st,not_ve_max_pck_width);
and_sen: and_gate port map (compt_send_increment,st,ve(max_pck_width));


compt_pck: compteur1 generic map ('0',max_pck_nbr+1,max_pck_width)
 port map (v1,ini,st);
compt_data: compteur1 generic map ('0',data_nbr+1,data_nbr_width)
 port map (v2,init,egal);
compt_receiv: compteur1 generic map ('1',rec_pck_nbr,rec_pck_nbr_width)
 port map (rec_pck,init,compt_receiv_increment);
compt_send: compteur1 generic map ('1',sen_pck_nbr,sen_pck_nbr_width)
 port map (sen_pck,init,compt_send_increment);
comparator: comparateur generic map (max_pck_width)
 port map (clk, init, egal, v1, ve (max_pck_width-1 downto 0));
memorie: rom1 generic map (data_nbr,max_pck_width+2,data_nbr_width)
port map (ve, v2);
F_S_M: fsm port map (res,st,ack_v_cp,req_v_cp,wr,clk,req_d_oprd,ack_d_cpu,req_d_cpu,
egal,init,ve(max_pck_width));
synch: oprd_synch generic map (nbr_req_sen,req_sen_width,nbr_ack_sen,ack_sen_width)
port map (ack_v_opr, req_v_opr, ve(max_pck_width+1), ve(max_pck_width), egal,
 req_d_oprd, ack_d_oprd, init,clk,req_demux_cmd, ack_demux_cmd);
rec_demux: demux generic map (rec_pck_nbr, rec_pck_nbr_width)
 port map (wr,rec_pck, wr_cmd);

in_out: mux_in_out generic map (com_bus_width) port map (ve(max_pck_width), com_bus,
 sign_to_oprd, sign_from_oprd);
l1: for i in 0 to rec_pck_nbr-1 generate
rec_regi: rec_reg generic map (com_bus_width)
 port map (clk,wr_cmd(i), data_out(((i+1)*com_bus_width-1) downto (i*com_bus_width) )
  sign_to_oprd);
end generate l1;

send_pack_mux: mux_generic generic map (sen_pck_nbr,com_bus_width,sen_pck_nbr_width)
 port map (data_in, sen_pck, sign_from_oprd);
req_demux: demux generic map (nbr_req_sen, req_sen_width)
 port map (req_v_opr,req_demux_cmd, req_v_oprd);
ack_demux: demux generic map (nbr_ack_sen, ack_sen_width)
 port map (ack_v_opr,ack_demux_cmd, ack_v_oprd);

end syn;
```